JAWAHARLAL NEHRU TECHNOLOGICAL UNIVERSITY - KAKINADA

C PROGRAMMING AND DATA STRUCTURES

UNIT - I

Algorithm / pseudo code, flowchart, program development steps, structure of C program, A Simple C program, identifiers, basic data types and sizes, Constants, variables, arithmetic, relational and logical operators, increment and decrement operators, conditional operator, bit-wise operators, assignment operators, expressions, type conversions, conditional expressions, precedence and order of evaluation.

Input-output statements, statements and blocks, if and switch statements, loops- while, do-while and for statements, break, continue, goto and labels, programming examples.

UNIT - II

Designing structured programs, Functions, basics, parameter passing, storage classes- extern, auto, register, static, scope rules, block structure, user defined functions, standard library functions, recursive functions, header files, C preprocessor, example c programs.

UNIT - III

Arrays- concepts, declaration, definition, accessing elements, storing elements, arrays and functions, two-dimensional and multi-dimensional arrays, applications of arrays. pointers- concepts, initialization of pointer variables, pointers and function arguments, address arithmetic, Character pointers and functions, pointers to pointers, pointers and multidimensional arrays, dynamic memory managements functions, command line arguments, c program examples.

UNIT - IV

Derived types- structures- declaration, definition and initialization of structures, accessing structures, nested structures, arrays of structures, structures and functions, pointers to structures, self referential structures, unions, typedef, bitfields, C program examples.

UNIT - V

Input and output – concept of a file, text files and binary files, streams, standard I/o, Formatted I/o, file I/o operations, error handling, C program examples.

UNIT - VI

Searching – Linear and binary search methods, sorting – Bubble sort, selection sort, Insertion sort, Quick sort, merge sort.

UNIT – VII

Introduction to data structures, singly linked lists, doubly linked lists, circular list, representing stacks and queues in C using arrays and linked lists, infix to post fix conversion, postfix expression evaluation.

UNIT - VIII

Trees- Binary tress, terminology, representation, traversals, graphs- terminology, representation, graph traversals (dfs & bfs)

TEXT BOOKS :

1. Computer science, A structured programming approach using C, B.A. Forouzan and R.F. Gilberg, Third edition, Thomson.
2. DataStructures Using C – A.S.Tanenbaum, Y. Langsam, and M.J. Augenstein, PHI/Pearson education.

REFERENCES :

1. C& Data structures – P. Padmanabham, B.S. Publications.
2. The C Programming Language, B.W. Kernighan, Dennis M.Ritchie, PHI/Pearson Education
3. C Programming with problem solving, J.A. Jones & K. Harrow, dreamtech Press
4. Programming in C – Stephen G. Kochan, III Edition, Pearson Eductaion.
5. Data Structures and Program Design in C, R.Kruse, C.L. Tondo, BP Leung, Shashi M, Second Edition, Pearson Education.

1 - INTRODUCTION TO COMPUTERS

1.1 WHAT IS A COMPUTER :

It is an electronic machine which accepts data as its input, process to it by doing some kind of manipulations and produce the output in a desired format. Technically a computer is a high speed electronic data processing machine.

1.2 DIFFERENCE BETWEEN CALCULATOR AND COMPUTER :

S.No	Calculator	Computer
1.	Works with numeric data	Works with alphanumeric data
2.	Cannot store information	Can store information or instructions
3.	Input in the form of pressing or touching push buttons.	Input may be in several forms, like : punch cards, magnetic tapes, joy sticks, etc.,

1.3 HISTORY AND EVOLUTION OF COMPUTERS :

2. **Abacus :** The first calculating machine ABACUS was developed by the Chinese. It is still being used in Russia and Japan.
2. **Analog Machine :** The Scottish mathematician, John Napier developed a device called Napier's bone analog machine around 1617. All arithmetic operations can be performed with the help of this machine.
3. **Odometer :** It is also called Speedometer. Along with basic arithmetic operations logarithms can be worked out.
4. **Mechanical Calculator :** The first mechanical calculator was made by the French Mathematician Blaize Pascal in 1642. It is a simple calculator used for addition and subtraction purposes. Later Leibnitz calculator was designed by Gottfried Leibnitz in 1671. It can be used for addition, subtraction, multiplication and division.
5. **Differential Engine :** Father of modern digital computers, Charless Babbage, an English Mathematician, proposed to build a general purpose problem solving machine, in 1833. This machine called as Analytical Engine was designed to perform all the five basic data processing operations namely input, storage, control, process and output.

1.4 COMPUTER GENERATIONS :

Over the years, many significant advances occurred in the computer industry based on the technology advances, the computer has been grouped into five generations.

First Generation : The first generation computers (1951-1958) used vacuum tubes technology. The first generation computers are heavy, occupying much space, unreliable and consumed large amount of power. It did not have the stored program concept.

- ENIAC (Electronic Numerical Integrator And Calculator) was the first computer made by J.P Eckert and J.W. Mauchly in 1946.
- EDSAC (Electronic Delay Storage Automatic Computer) was designed by M.N. Wilkes in 1949.
- EDVAC (Electronic Discrete Variable Automatic Computer) was designed by Pennsylvania in 1950.
- UNIVAC (Universal Automatic Computer) was made by universal Accounting Company set up by Eckert and Mauchly in 1951.
- IBM-701 & IBM-650 : Were introduced in 1953 and 1954 respectively by International Business Machine corporation (IBM). IBM-650 was the first modern digital computer.

Second Generation : Second generation computers (1959-1964) used transistors technology. The size of transistor is $1/200^{th}$ size of vacuum tube. Transistor helped in developing smaller and more reliable computers which used less power and generated less heat than the first generation.

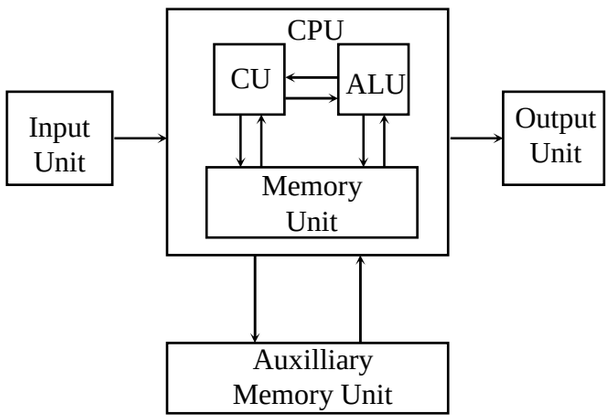
Third Generation : The third generation computer (1964-1970) was characterized by IC (Integrated Circuit) technology, which contained many components fused together on a single chip. These computers are very reliable and could store programs. The cost of these machine has come down making them available to business. Many programming languages and operating systems were developed for use on these computers. These computers were also called as mini computers.

The IC's are classified into 4 types; these are SSI (Small Scale Integrated Circuit), MSI (Medium Scale Integrated Circuit), LSI (Large Scale Integrated Circuit), and VLSI (Very Large Scale Integrated Circuit).

Fourth Generation : To increase the speed, greater reliability, large storage capacity etc., fourth generation computers were developed from 1971 onwards. These computers are developed by using Microprocessor technology, Hence these computers are called as Micro Computers or Micros.

Fifth Generation : In this generation, AI (Artificial Intelligence) technology is used.

1.5 BLOCK DIAGRAM OF A COMPUTER:



The computer consists of four blocks, they are :

1. Input Unit
2. Central Processing Unit
3. Output Unit
4. Auxiliary Memory Unit

Input Unit : Input unit is provided for man-to-machine communication. It accepts data in human readable form, converts it into machine readable form and sends it to CPU. A computer may have one or more input devices, depending upon its type, size and use. Keyboard and mouse are most commonly used input devices. Other input devices are : Punch card reader, Punched paper tape reader, Optical scanners, Magnetic Ink Character Reader (MICR), Voice Data Entry terminal (VDE), etc.,

Central Processing Unit : It consists of three units, Control Unit (CU), Arithmetic Logic Unit (ALU) and Memory Unit (MU). The main function of CPU is :

1. Control the sequence of operation as per the stored instructions
2. Issue commands to all parts of the computer
3. Stores data and instructions
4. Process the data and sends results to output.

Control Unit : The control unit controls and co-ordinates all operations of the CPU, Input and Output devices.

1. It gives commands to transfer data from input unit to memory unit, and arithmetic logic unit.
2. It stores the program in the memory, accesses instructions one by one, issues appropriate commands to other units according the instructions.
3. It transfers the results from ALU to the memory unit and output unit.
4. It fetches the required instructions from the main memory, and interpret it by sending appropriate signals to the concerned hardware device.

Arithmetic and Logic Unit : It carries out all arithmetic operations like addition, subtraction, multiplication and division. Also it performs all logical operations.

Memory Unit : It is used to store programs and data. It is mainly two types, they are :

1. Main memory or Primary memory or Immediate Access Storage (IAS), which is part of the CPU.
2. Auxiliary memory or Secondary storage, which is external to the CPU.

Output Unit : Output unit is provided for machine-to-man communication. It receives the information from CPU in machine readable form and presents it to the user in a desired form. A computer may have one or more output devices depending upon use. The Visual Display Unit (VDU), and Printers are most commonly used devices.

1.6 I/O DEVICES :

The computer communicates with the outside world through input and output devices, i.e. man-to-machine and machine-to-man communication. These

input/output devices are usually referred as I/O devices and some times as peripheral devices.

Input Dvices : The input devices are broadly classified as Text entry and Non-text entry devices. The text entry devices are used when exclusively text and numeric values are present in the input data. The non-text entry devices are used to enter the values that used to be translated into machine readable form. The system may also support one or more of the following.

Punched Card: Few years back punched cards were the most widely used device. It is 80 column card and each column represents one character so a maximum of 80 characters can represent in a single card. This coding system used to represent data is known as Hollerith code. A card punch machine is used to write data into cards. It looks like a type writer which punched a set of holes in a card instead of printing character on a paper. Each character is represented by a unique set of holes according to the hollerith code. It has advantages like : (i) easy to read data (ii) error in the data can be changed by siply replacing the wrong card with the right one. It has disadvantages like : (i) transfer of data is very slow (ii) cards are easily damaged.

Paper Tape : There have a series of column with characters represented by a pattern of holes. It has advantages : (i) compared to punch cards it is more compact and economic (ii) It is very ligt and easy for handling. It has disadvantages like : (i) The role is so long correcting or inserting data can be very tedious (ii) paper life is short.

OMR (Optical Mark Recognition) : In this method special pre printed forms are designed with boxes can be marked with a dark pencil or ink. The optimal mark reader uses a light source which sense the presence of mark which transcribed the marks into electronic pulses. Nowadays, it is quite a common practice to judge the knowledge and IQ level of a candidate by multiple choice objective type questions. The candidates are required to write their answers by putting a standard dot in front of the correct answer with soft pencil. OMR can read about 10,000 documents per hour.

OCR (Optical Character Recognition) : OCR units scan the sheets of paper on which the data is present and detect the alphabetic and numeric characters. These can detect printed, typed and hand written data. OCR units consist of high source detectors, they detect and visualize each character in a pattern of light and dark images. The light source detectors move across the page and detect the characters and result the character string. A wide variety of fonts can be detected by these units. The system tends to make errors when the paper is dirty, greased, or poorly typed. Hence quality paper should be used with OCR units.

MICR (Magnetic Ink Character Reorganition) : This is used in the banking industry in processing large volume of checks. The bank's identification code and the customer's account number is pre printed on these checks with a special ink that contains magnetizable particles of iron oxide. This system is highly reliable and also ensures accuracy and saves time.

Bar code Recognition : This method is generally used in printing the price of goods. The prices are coded in the form of lines of bars of varying thickness. The series of such bars encodes the information of product. Bar code reading may be either done by physical contact using light pen which is linked to the computer or it may be done by scanning laser. There are several bar code systems in use. Some transmit only numeric information while others permit complete ASCII characters.

Keyboard : It is most commonly used input device. It is similar to the face of type writer. It contains letters A to Z, numbers 0 to 9 and punctuation marks used for normal text. Inaddition to this it contains some special functional keys (F1, F2, etc.,) which are used as extensive commands in special programs. Also it has some special purpose keys like CTRL, ALT, ESE, spacebar, etc., When any character is entered by the user the corresponding numeric code or its ASCII equivalent is normally transmitted by the keyboard.

Basically a keyboard can be divided into three regions. At the first region generally on the top side of the keyboard contains functional keys. In the second region the keys are similar to the normal type writer keys and some of the special keys. The third region on the right hand side of the keyboard consists of numeric pad which allows the programmer to enter the numeric data at a faster. The arrow marked keys control the cursor movements.

Mouse : It is used to position the cursor on the screen. It is a small palm sized box. Its manipulations on a flat surface move the cursor in the same direction as the movement of the mouse. It is also the most commonly used input device.

Joystick and Trackball : Children can play games on computers in a simple way by the use of Joystic or Tackball. It does not require much space on your desktop.

The ball moves as you rotate it with your thumb, your fingers, or the palm of your hand. Trackballs come in various sizes and shapes.

Light Pen : You move the pointer and select items on the display screen by directly pointing to the objects on the screen with the pen.

Touch Screen : It allows you to use your finger to touch the screen and choose objects. A special computer monitor or a monitor fitted with special hardware is necessary to provide touch screen capabilities.

Pointing Sticks & Touch Pads : Pointing sticks are the short red sticks that are positioned in the middle of the keyboard on some laptop computers. These devices are pressure-and-direction sensitive, so the cursor moves in the direction that the pointer is pressed; the cursor moves faster when more pressure is applied. Touch pads can be found in both laptops and desktop computers. By moving and tapping your finger on the pad, you can relocate the pointer on the display screen and simulate a mouse click. Touch pads can be valuable for computer users dealing with hand coordination problems.

CD-ROM (Compact Disk Read Only Memory) : Used as an input device. Nowadays, we are getting a variety of CD-ROMs. Mainly used for massive data storage.

Output Devices : The output devices receive information from the CPU and present it to the user in a desired form. Printers and VDUs are most commonly used output devices.

Display Screen (or) Visual Display Unit (or) Monitor : When a text is keyed in, the screen displays the characters. The user can read the text line by line and make corrections before it is stored or printed on a printer. A VDU screen in made up of Cathode Ray Tube(CRT) due to which it is also called CRT terminal. The VDU's are available in two forms.

1) **Monochrome** : These are black and white, green phosphor, or amber monitor. They do not receive any TV signals.
2) **Coloured** : Display is in colour, and these are clearer and sharper than the monochrome monitors. There are 24 lines on one screen and each line contains 80 characters. Any picture or diagram is shown by a combination of 640 columns and 200 rows of tiny dots. These tiny dots that can be shown on the screen are called pixel (picture element)

Printers : The final output obtained from a printer is often referred to as a printout. Printers are available with a variety of printing mechanism such as speed and varying quality. The common categories of printers available in the market are impact printers and non-impact printers.

1) **Impact printers** : Impact printers print by striking the type against the ribbon and the paper, similar to the type writer. They can produce carbon copies if necessary
2) **Non-impact printers** : Ink-jet printers are examples of this type. These printers print characters by spraying charged ink on to the paper. They are capable of producing a variety of character shapes and sizes. The speed is around 100 cps. Though they can produce good quality characters of different styles and sizes and print even logos for letter heads, they are very expensive, and therefore, are not popular.

Character printers : Printers that can print only one character at a time are called character printers. These are basically of two types.

1) **Dot Matrix Printer** : These are inexpensive and the speed varies from 40 to 6000 cps (character per second) and they produce a variety of character sets. The main disadvantage with these printers is the poor print quality.
2) **Daisy Wheel Printers** : These printers give very high quality print. These are slow, noisy and more expensive than dot matrix printers.

Line printers: The line printers, print an entire line at a time rather than a character. It is the fastest printer. Its speed varies from 300 to 3000 lines per minute. Line printers generally produce output only in upper case letters though some have provision for both upper and lower case letters. The quality of printing is poor. These are used for producing large volume of reports in a big organizations. These are the two types of line printers namely drum type printer and chain type printer.

Page printers (or) Laser printer : The output of these printers is better than the character and line printers. These printers can print characters of different sizes and styles; they can even be used for printing pictures.

Plotters (Tracers) : Plotters are used to produce output containing graphics or diagrams. Multi colour plotters are used for preparing financial documents, annual repots and engineering drawings. Plotters may either use pen or ink-jet approach. Pen plotters are available in two forms, drum type and flat bed type. In the drum

plotters both paper and the pen move. The ink-jet plotters use jets of ink with different colours and are able to produce large drawings containing many colours.

1.7 MEMORY :

Instructions and data have to be stored at a place in the computer till they are needed. The data and programs are stored in the computer at 3 different levels.

1. Main memory or Primary memory or Immediate Access Storage (IAS)
2. Auxiliary memory or secondary memory
3. Cache Memory.

Main Memory : It is also called as primary memory/main memory/Immediate access storage, and it is available inside the CPU. Data that is just entered and data that is currently being used is available in this memory. It is a fast memory, made up of a large number of cells to store a fixed capacity of storage and have unique address. Different types of primary memory are :

1. **RAM** (Random Access Memory) : All the data entered into the system is directly stored in RAM. The user has direct access to this part of memory i.e., the user can read and can write into this memory. Hence this memory is also known as Read/Write memory. The contents of this memory are not permanent because once the system is switched off or power goes the contents are erased, hence it is volatile memory. RAM is of two different types
   a) Dynamic RAM : It is volatile memory which stores information temporarily. It is constructed by using the charging and discharging of capacitors.
   b) Static RAM : It is non-volatile memory constructed by using semi-conductor materials. This RAM uses semiconductors, transistors, diodes for its construction.
2. **ROM** (Read Only Memory) : In this part of memory some instructions are permanently loaded during the manufacturing of the computer. These instructions are globally used by the user. No changes can be done in this memory.
3. **PROM** (Programmable Read Only Memory) : In this part of the memory the user can put the data according to his specification using a special device known as prom-programmer. However once the chip has been programmed, the recorded information cannot be changed. This is non-volatile memory.
4. **EPROM** (Erasable Programmable Read Only Memory) : It is similar to PROM, but erasing can take place by exposure to ultra violet light.
5. **EEPROM** (Electrically Erasable Programmable Read Only Memory): EEPROM can be easily reprogrammed by the application of small voltage.

1.8 SECONDARY MEMORY :

It is also called as External Storage Devices or Secondary Storage Devices : The main purpose of the external storage is to retain data and programs for further use. If information is stored in an external storage media then the operator can retrieve it as and when required, thus avoid repeat typing. It is non-volatile memory. The popular external storage devices are:

1. **Magnetic Tapes :** It provides serial access. Therefore you have to read all the previous records to reach a particular records. Information can be erased by recording new information in its place. The tape has a ferromagnetic coating on a plastic base and is similar to the tape used in tape recorders. Information recorded on magnetic tape is stored in varying densities, typically 1600 CPI or even as dense as 6250 CPI.
2. **Magnetic Disks :** It allows direct access but can also be used in serial mode, if required. In shape a disk resembles an L P Gramophone record. Each disk consists of a number of invisible concentric circles called tracks. Each track further is divided into sectors. Information is recorded on the tracks of a disk surface in the form of invisible tiny magnetic spots. Magnetic disks are of two types :
   1) **Floppy disk :** This is the most commonly used storage medium on personal computers. Information can be recorded or read by inserting it into a disk drive connected to the computer. The disks are permanently encased in stiff paper jackets for protection and easy handling.
   2) **Hard disk** : It is used for storing large volume of information. Popularly known as Winchester disk. These are very easy to read and write and store compared to the floppy disk. But the disadvantage is it cannot make backup copies and is also not transportable and expensive as compared to floppies. Also it cannot be removed from drives.

Disk Drives : The mechanism that drives the magnetic disks is called the disk drive. In this case of Winchester disks the drive mechanism and the disks are integrated together to form the Winchester drive. However the floppy drives are separate; the

floppies can be inserted into drives as and when it is required for use. The disks spin at a high speed under a read/write head.

Cache Memory : Cache memory is a high speed storage, i.e., much faster than the main memory. Cache memory is highly expensive compared to the main memory. Programs in the main memory are shuffled to the very high speed cache before being executed.

1.9 BITS, BYTES AND WORDS :

A bit (Binary digit) is the smallest unit of information, i.e. either 0 or 1. The size of memory is specified in terms of bytes. A Byte is made of 8 bits. A Word may be formed by using two of more bytes. The basic information unit of computer is called a word.

1 Binary digit	---	1 bit
4 Bits	---	1 Nibble
8 Bits (2 Nibbles)	---	1 Byte
16 Bits (2 Bytes)	---	1 Half Word
32 Bits (4 Bytes)	---	1 Full Word
64 Bits (8 Bytes)	---	1 Double Word
1024 B	---	1 KB (Kilo Byte)
1024 KB	---	1 MB (Mega Byte)
1024 MB	---	1 GB (Giga Byte)

1.10 BINARY CODES :

In computers code conversion is not much suitable for performing I/O operations. So to overcome this disadvantage binary codes were introducted. The binary codes can be broadly classified into two types : 1. Numeric codes, 2. Alphanumeric codes.

1. **Numeric Codes :** When 0 – 9 decimal digits are coded in binary using a sequence of 0s and 1s, the system is called as a numeric coding system. BCD is the best example for Binary codes.
2. **Alphanumeric Codes :** When 0 – 9 decimal digits, A – Z English upper case letters, a – z English lower case letters, (+, -, *, /, ^, etc., arithmetic operators, <, >, =, etc., relational operators, ,, ., :, ;, “, ‘), etc., separators, [ ], { }, ( ) etc., parenthesis, CTRL, ALT, etc., special characters are given codes in binary using a sequence of 0s and 1s, then the coding system can be called alphanumeric coding system. The relevant examples for alphanumeric coding system are 1. **ASCII** (American Standard Code for Information Interchange) (pronounced as ask-ee) It is an 7 bit code, and ranges from 0 to 127. 2. **EBCDIC** (Extended Binary Coded Decimal Interchange Code) (pronounced as sa-ed-si-dik) is an 8 bit code. The major application of binary codes is input, output and storage operations. 3. **Unicode** defines a fully international character set that can represent all the characters found in all human languages. It is a unification of dozens of character sets such as Latin, Greek, Arabic, Cyrillic, etc. Generally Unicode characters are represented by escape sequences. It is a 16-bit (2 Bytes) code; it ranges from 0 to 65,536.

1.11 CLASSIFICATION OF COMPUTERS :

Computers are broadly classified into three types. They are Analog Computers, Digital Computers and Hybrid Computers.

Analog Computers : Analog Computers use an analog signal for its operation. These computers work on the basic principle of measuring of signal. These computers are very accurate, but the cost increases directly in proportion to its accuracy. Example : Speedometer.

Digital Computers : Digital computers work with digital signal. These computers work using 0s and 1s. The basic principle is counting of digits. These computers can perform a variety of operations. Digital computes are inexpensive and comparatively less approximate.

Hybrid Computers : A hybrid computer is a combination of both digital and analog computers. These computers exhibit both the good features of digital and analog computers. These computers are used for special applications such as industrial auto machine.

1.12 TYPES OF DIGITAL COMPUTERS :

1. **Super Computers :** Super Computers are most powerful computers. These computers have the capability of around 64 mainframe computers. At a time 120 users can interact with it. These computers are generally used for weather forecasting, missile firing, satellite launching etc., These computers are also called as Maxi Computers and Monster Computers.

2. **Mainframe Computers :** A mainframe computer is a large computer having the capability of around 16 mini computers. At a time 64 users can interact with these computers. These computers are generally used in business and scientific applications. These computers are most commonly used for problems such as weather forecasting. These computers are also called as Midi Computers.
3. **Mini Computers:** A mini computer is 16 times faster than a micro computer. At a time 12 users can interact with these computers. Mini computers are commonly used for business and scientific applications.
4. **Micro Computers :** A micro computer is the smallest computer, which uses a micro processor as its CPU. A Micro Computer also called as Personal Computer (PC), can interact with single user. These computers are inexpensive and are popular. A micro computer will be used for industrial, business and scientific applications. Personal Computers are classified into PCs, PC-XTs (Personal Computer Extended Technology) and PC-ATs (Personal Computer Advanced Technology).

Difference Between PC, PC-XT and PC-AT :

	PC	PC-XT	PC-AT
Processor	8088	8088	80286 onwards
Storage	FD	FD/HD	FD/HD
Speed	8 MHz	8 MHz	>= 12 MHz
Category	Single User	Single User	Multi User

1.13 COMPUTER PROGRAMMING LANGUAGES :

Computer programming means giving instructions to a computer, in the language which computer understands. Instructions can be written in two different languages. They are 1. Low Level Language, 2. High Level Language.

Low Level Language : These are basically two types, they are : 1. Machine Language, 2. Assembly Language.

1. **Machine Language :** A computer is an electronic machine which can understand any instruction given to it in a coded form. Since its memory can store only 0s and 1s, the instruction must be given to computer in binary code. A program written in binary coded form (or any other coded form like Hexa, Octal, etc.,) that can be directly fed into the computer for execution is known as machine Language. Machine language differs from processor to processor. Writing a program in machine language is a very difficult and cumbersome process. Machine language contains instructions written in very long numerical chains and it is very difficult to remember the code.
2. **Assembly Language :** It is a symbolical language to reduce the programmers burden, developed in 1950s. This language permits the use of alphanumeric symbols (numbers and letters) instead of numeric operations. An assembly language program contains mnemonics because it is also called mnemonical language. Mnemonic means that the information can be memorized easily and is generally in the form of abbreviations. Assembly language is not cumbersome to use as the machine language. It is machine oriented.

Machine language (Vs) Assembly language :

1. Execution of machine language program is faster than the assembly language program, because there is no conversion in machine language.
2. Reading and writing the assembly language programs are easy for programmers, it is difficult in machine language.
3. Machine instructions are difficult to remember, where as mnemonics are easy to remember.
4. Introduction of data to program is easier in Assembly language, while it is difficult in machine language.

High-Level Languages : A set of languages developed which are very close to our native languages, are called High-Level languages. High level languages were designed to provide high level control structures, Input/Output facilities, hardware independents and so on. A single instruction or statement in a high level language produces a number of machine language instructions. High level languages can be put into four general categories. They are : (i) Scientific languages (ii) Commercial languages (iii) Special purpose languages (iv) Multipurpose languages. Ex. : BASIC, COBOL, FORTRAN, PASCAL. C

Advantages :

1. Easy to learn and easy to understand than machine or assembly languages.
2. Takes less time to write.
3. Easy to maintain.
4. Better documentation

1.14 HARDWARE :

It is a term used to represent the physical and tangible components of the computer itself i.e. those components which can be seen and touched. Input, Output, Memory devices, CPU are the examples for computer hardware.

1.15 SOFTWARE :

The physical equipment that makes up computer will not work unless there are instructions to guide the device. These instructions that are grouped as a set is called a program. These set of programs are called software. The hardware cannot function without software and software is created by people.

Types of softwares : Software can be classified into two types. They are system software and application software. Both types of software is necessary for the computers.

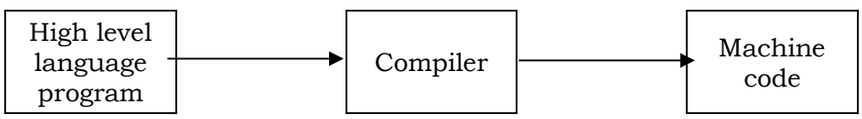
System Software : System software includes all routines that inside the computer memory and helps the user to write or execute application programs. This facilitates the communication between user and computer. This includes Assembler, Compiler, Interpreter, Editors, etc.,

Application Software : Application programs are written for solving a specific problem. Sales analysis, word processing, billing etc., are examples of application softwares.

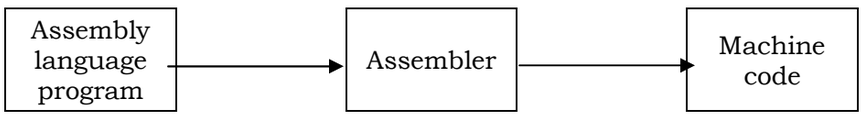
1.16 TRANSLATORS :

Translators are software that accepts one language as input and converts it into another language. The input for any translator is called as source language or source code and output of any translator is called as object language or object code. There are two type of translators; they are :

1. **Compiler :** It is a software, that accepts the high level language program as input and produces the machine code as output.



2. **Assembler :** It is a software that accepts the assembly language program as input and produces the machine code as output.



1.17 OPERATING SYSTEM :

It is a system software, which controls the execution of computer programs and which may be provide scheduling, debugging, Input/Output control, accounting, compilation, storage assignment, data management and related services. Any operating system is responsible for : 1. Memory management, 2. CPU Management, 3. Device Management, 4. I/O device Management, 5. File Management and 6. User Interface. The operating system provides an interface between hardware and the user programs. The operating system is the first program run on a computer when the computer boots up.

2 - INTRODUCTION TO 'C'

2.1 ALGORITHM :

An algorithm is the step-by-step logical procedure for solving a problem. Each step is called an instruction. An algorithm can be written in English like sentences or in any standard representation. The algorithm written in English like language is called '***pseudo code***'. All algorithms must satisfy the following properties.

1. **Input:** Zero or more inputs.
2. **Output:** At least one quantity is produced.
3. **Definiteness:** Each instruction is clear and unambiguous.
4. **Finiteness:** The algorithm should terminate after a finite number of steps. It should not enter into an infinite loop.
5. **Effectiveness:** Each operation must be simple and should complete in a finite time.

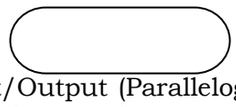
A problem can have several algorithms. Each algorithm represents a different way of solving the given problem. For example, there are several different algorithms for sorting the n numbers. In such a situation we would like to pick up the best or most efficient algorithm for the given problem. Therefore we need a way of characterizing the efficiency of an algorithm. Time complexity is one way of characterizing the efficiency or work to be done by an algorithm.

2.2 FLOWCHART :

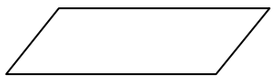
A flowchart is a pictorial representation of an algorithm. It shows the flow of operations in pictorial form and any error in the logic of the problem can be detected very easily. A flowchart uses different shapes of boxes and symbols to denote different types of instructions. These symbols are connected by solid lines with arrow marks to indicate the operation flow. Normally an algorithm is represented in the form of a flowchart and the flowchart is then expressed in some programming language to prepare a computer program.

Flowchart Symbols : A few symbols are needed to indicate the necessary operations in a flowchart. These symbols have been standardized by the American National Standard Institute (ANSI). These symbols are :

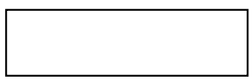
1. **Terminal :** The terminal (Oval) symbol as the name implies is used to indicate the beginning <START>, ending <STOP> and pause <HALT> in the program logic flow. It is the first and last symbol in the program logic.



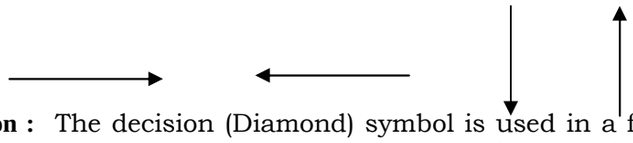
2. **Input/Output :** The Input/Output (Parallelogram) symbol is used to denote any function of an Input/Output device in the program. All input output instructions in the program are indicated with this symbol.



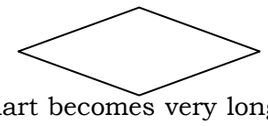
3. **Processing :** A processing (Rectangle) symbol is used in a flow chart to represent arithmetic and data movement instructions. Thus all arithmetic processes of addition, subtraction, multiplication and division are shown by a processing symbol. The logical process of moving data from one location to another in the main memory is also denoted by this symbol.



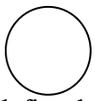
4. **Flow lines :** Flow lines with arrow heads are used to indicate the flow of operations i.e. the exact sequence in which the instructions are to be executed. The normal flow of flowchart is from top to bottom and left to right. Arrow heads are required only when the normal top to bottom flow is not to be followed. However as a good practice and in order to avoid ambiguity flow lines are usually drawn with an arrow head at the point of entry to a symbol.



5. **Decision :** The decision (Diamond) symbol is used in a flowchart to indicate a point at which a decision has to be made and a branch to one of two or more alternate points is possible.

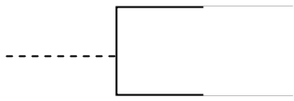


6. **Connectors :** If a flowchart becomes very long the flow lines start criss cossing at many places that causes confusion and reduces understandability of the flowchart. Moreover there are instances when a flowchart becomes too long to fit in a single page and the use of flow lines become impossible. A connector symbol is represented by a circle. A pair of identically labelled connector symbols is commonly used to indicate a continued flow when the use of a line is confusing. So two connectors with identical labels can be used.



7. **Predefined Process :** The predefined process (Double sided rectangle) symbol is used in flowcharts to indicate that modules or subroutines are specified elsewhere.

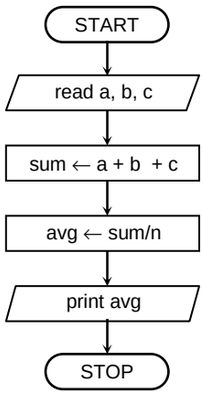
8. **Annotation :** The annotation (Bracket with broken line) symbol is used in flowcharts to indicate the descriptive comments or explanation of the instruction.



2.3 EXAMPLES ON FLOW CHARTS AND ALGORITHMS :

1. Write an algorithm for finding the average of given three numbers and draw the flowchart for the same :

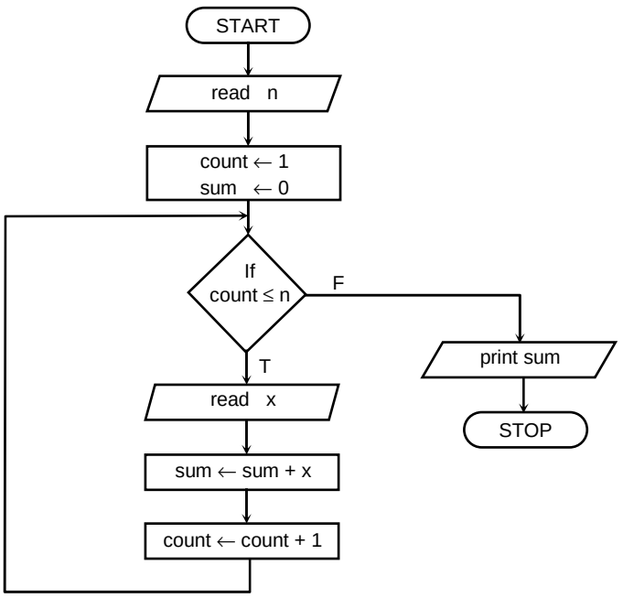
[Step 1]: [Read the values]
Read a, b and c
[Step 2]: [Compute]
sum = a + b + c
avg = sum/3
[Step 4]: [Display output]
Print avg
[Step 5): [End]
Stop



2. Write an algorithm for finding the sum of given n number and draw the flowchart for the same :

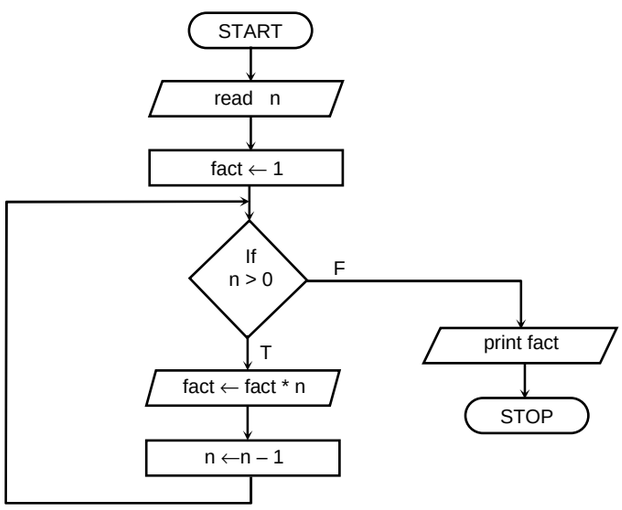
[Step 1]: [Read the values]
read n
[Step 2]: [Initialization]
count = 1
sum = 0
[Step 3]: [Perform the loop]
repeat through step 4 while count <= n
[Step 4]: [Read and Compute]
read x
sum=sum+x
count=count+1
[Step 5]: [Display output]

print sum
[Step 6]: [Finished]
Stop



3. Write an algorithm for finding the factorial of a given number and draw flowchart for the same :

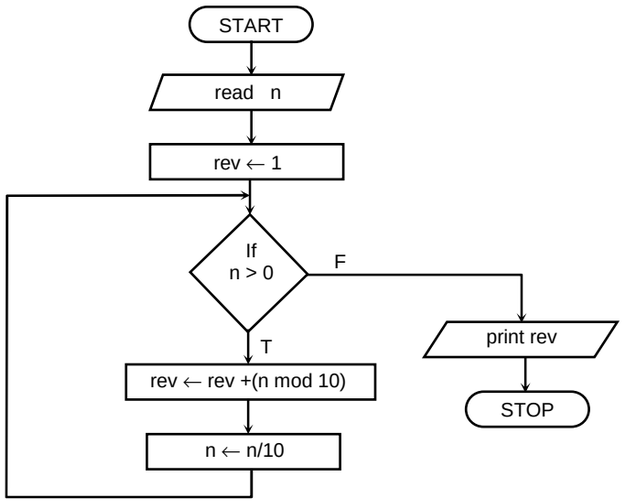
[Step 1]: [Read the values]
read n
[Step 2]: [Initialization]
count=1
fact=1
[Step 3]: [Perform the loop]
repeat through step 4 while count <= n
[Step 4]: [Compute]
fact=fact*count
count=count+1
[Step 5]: [Display output]
print fact
[Step 6]: [Finished]
Stop



4. Write an algorithm for finding the reverse of a given number and draw the flowchart for the same :

[Step 1]: [Read the values]
read n
[Step 2]: [Initialization]
rev=0

[Step 3]: [Perform the loop]
repeat through step 4 while n>0
[Step 4]: [Compute]
rev=rev*10+n mod 10
n=n/10
[Step 5]: [Display output]
Print rev
[Step 8]: [Finished]
Stop



2.4 PROGRAM DEVELOPMENT STEPS :

Generally programmers can be broadly categorized as two types such as successful and nonsuccessful programmer. The second category programmers rush to their keyboards and begin coding, i.e. the actual writing of instructions. But its is the last stages task for the successful programmers. There are many more important steps to consider first.

Understand the problem : Unless the problem is clearly understood, you cannot even begin to solve it. This seems like a truism until you appreciate that a program specification seldom gives all the facts required by the programmer. The professional programmer is a pessimist, because from past experience there is always some importance information which is omitted. This needs to be identified first.

Examine the data : Programs are written to work on data. Unless one knows exactly how the data is organized, what it 'looks' like, etc., the program which processes it cannot be written. This fact becomes clearer the more one writes programs, but it is a fact all too frequently overlooked by the novice.

Plan the output : The output should be planned next. Not only does this help to ensure that nothing is omitted from the program, but it helps to get a clear picture of what the program is trying to achieve and whether the programmer really does understand the problem.

Designing the solution : There are many ways of designing solution, so much so that entire books are devoted to this subject alone. Computer scientists frequently say that programming is like any engineering task in that the program has to be designed and constructed in much the same way as any engineering project. A motorway is not built by starting at point A and steadfastly pushing on to point X. Rather, months are spent in planning; charts designed; sub-tasks identified as well as those which cannot begin until others have been completed; broad designs are developed and later more detailed designs constructed. It is only after a long planning period and the most effective order of the sub-tasks is agreed upon that the construction crew actually begins the work. Programming requires the same painstaking processes, with the end result standing or falling by the amount of care and attention invested in the planning stage.

Selecting test data : How can one ensure that once a program is eventually working the results it produces are 'correct'? The answer is simple common sense. Try the program out on some data to which the answers have been worked out in advance. If they match, the program should be all right. Selecting effective test data is a serious exercise and the more significant the program, the more care needs to be taken in the selection.

The actual coding : At this stage, one can begin to code the detailed program designs into program instructions of a given language. If all the previous steps have been completed with due diligence, this coding should be almost 'automatic'. The

chances are high that a fairly successful program will result first time around. Although it may still contain bugs (errors), these should be fewer and relatively easy to identify and correct.

Testing : The program can be tested with the test data, results checked and any errors amended. When all is correct, the program can be released and set to work on live data.

2.5 HISTORY AND EVOLUTION OF C :

'C' seems a strange name for a programming language, but it is one of the most popular programming languages today. 'C' was originally developed in the 1970's by Dennis Ritchie at Bell Telephone Laboratories Inc. 'C' was an offspring of the BCPL (Basic Combined Programming Language) called B. The C language is often described as a "Middle-level" language, because it combines the best features of high level languages with the control and flexibility of assembly language. C is a general purpose structured programming language that has much in common with the best of the high-level language.

Features and Applications of 'C' language :

1. 'C' is general purpose structured programming language.
2. 'C' is powerful efficient, compact and flexible.
3. 'C' is highly portable.
4. 'C' is a robust language, whose rich set of built-in functions and operators can be used to write any program.
5. 'C' is well suited for writing system software, as well as application programming.
6. 'C' has the ability to extend itself. We can continuously add our own functions to the existing 'C' library functions.
7. 'C' programs can be run on any of the different computers with little or no alteration.
8. 'C' is widely available Commercial. 'C' compilers are available on most personal computers, mini and mainframes.
9. 'C' language allows reference to a memory location with the help of pointer which holds the address of the memory location.
10. 'C' language allows dynamic allocation of memory i.e. a program can request the operating system to allocate/release memory.
11. 'C' language allows manipulations of data at the lowest level i.e. bit level manipulation. This feature is extensively useful in writing system software programs.
12. C is a case sensitive language

2.6 BASIC STRUCTURE OF 'C' PROGRAM :

A 'C' program is a group of building blocks called functions. A function is a subroutine that may include one or more statements designed to perform a specific task. To write a C program, we first create functions and then put them together. Any C program may contain one or more sections as shown in the below.

Documentation Section	// Optional
Link Section	// Optional
Definition Section	// Optional
Global declaration Section	// Optional

main() function Section
{
Declaration Part
Executable Part
}
	// Must

Sub Program Section Function 1 Function 2 . . Function n	// Optional

Document Section: This section consisting of comments specifies the name of the program, author and other details. These comments beginning with the two characters /* and ending with the characters */. No space can be included between these pairs of characters (delimiters), any characters may be included in either uppercase or lowercase.

Link Section: This section provides the compiler to link functions from the system library (Ex: stdio.h, math.h etc.,).

Definition Section: This section defines all symbolic constants.

Global Declaration Section: Some variables are used in one or more function; such variables are called Global variables and are declared in this section i.e. outside of all the functions.

Main function section: Every C program must have atleast one function i.e. main() function. This main section have two parts (1) declaration part and (2) execution part. The declaration part declares all the variables that are used in the executable part. There is at least one statement in the executable part. These two parts can appear between the opening and closing braces. In all C programs execution begins at this opening brace and ends at this closing brace. The closing brace of this function section is the logical end of the program. All the declaration and executable statements end with a semicolon.

Subprogram section: This section contains user defined functions that are mentioned in the main function. User defined functions are generally placed immediately after the main function.

All sections, except the main function section may be absent when they are not required.

2.7 A SAMPLE 'C' PROGRAM :

This is a example program to print a string.

```
1.  /* A sample C program */
2.  main()
3.  {
4.  /* prints the string */
5.  printf("Welcome to C world \n");
6.  }
```

In the above program first and fourth lines are commented lines. These are used in a program to enhance its readability and understanding. Comment lines are not executable statements i.e. anything between /* and */ is ignored by the compiler. These comment lines can be inserted wherever we want them in the program.

The second line informs the system that the name of the function is ***main*** and the execution begins at this line. The ***main()*** is a special function used in the C language to tell the computer where the program starts. The empty pair of parentheses immediately following ***main*** indicates that the function has no arguments.

In the third line the opening brace "{" indicates the beginning of the function ***main*** and in the last line the closing brace "}" indicates the end of the function. All the statements between these two braces is the function body.

Here the function body consists of two statements, one is comment line and another one is ***printf*** line is an executable statement. The ***printf*** is a predefined function to print the characters which appear in between quotation marks, here the output will be "Welcome to C World".

2.8 EXECUTING A 'C' PROGRAM UNDER MS-DOS SYSTEM :

C program execution involves the following series of steps.

1. **Creating the program:** The text of a program is called the source code, a sequence of statements to be executed by the machine. This source code is usually stored in files with the extension of **.**c.
2. **Compiling and linking of the program:** Before a program is made to run, it must be translated by the compiler to give .OBJ files and then linked by the linker to give .EXE file.
3. **Executing the program:** The extension for executable codes is .EXE. A 'C' program with an extension .EXE can be run in DOS prompt mode.

2.9 OBJECTIVE TYPE QUESTIONS :

1. Who developed the C language? [ ]
   a) Ken Thompson b) Bjarne Stroustrup
   c) Dennis Ritchie d) Kernighan
2. When was the C language developed? [ ]
   a) 1970 b) 1972 c) 1975 d) 1976
3. Algorithm written in English like language is called _____ [ ]
   a) Object Code b) Pseudo Code c) Source Code d) None
4. _____ is a pictorial representation of an algorithm. [ ]
   a) Program b) Language c) Flow chart d) None
5. Flow chart symbols are standardized by _____ [ ]
   a) ASCII b) ANSI c) IBM d) None
6. _____ symbol is used to indicate a pause <HALT> [ ]
   a) Oval b) Parallelogram c) Rectangle d) Circle
7. _____ symbol is used to represent arithmetic operation [ ]
   a) Oval b) Parallelogram c) Rectangle d) Circle
8. The logical processing of moving data from one location of the main memory to another is denoted by _____ symbol. [ ]
   a) Oval b) Parallelogram c) Rectangle d) Circle
9. Double sided rectangle is used to represent _____ [ ]
   a) Predefined process b) Input/Output
   c) Computing d) None
10. _____ symbol is used to represent the comments [ ]
    a) Bracket with broken line b)Parallelogram
    c) Rectangle d) Circle
11. A compiler compiles the source code [ ]
    a) One line at a time b) Complete program in one stroke
    c) Two lines at a time d) Five lines at a time
12. An interpreter reads the source code of a program [ ]
    a) One line at a time b) Complete program in one stroke
    c) Two lines at a time d) Five lines at a time
13. The C language is closely associated with _____ operating system [ ]
    a) MS-DOS b) Unix c) MS-Windows d) Linux
14. The extension for C program files by default is [ ]
    a) 'cpp' b) 'exe' c) 'obj' d) '.c'
15. C program should be written in [ ]
    a) Uppercase b) Lowercase
    c) Titlecase d) None of the above
16. C is a [ ]
    a) High-level language b) Middle-level language
    c) Low-level language d) Machine language
17. Each statement in a C program is terminated with [ ]
    a) Period(.) b) Slash(/) c) Semicolon(;) d) Hash(#)

ANSWERS

1.	c	2.	b	3.	b	4.	c	5.	b
6.	a	7.	c	8.	c	9.	a	10.	a
11.	b	12.	a	13.	b	14.	d	15.	b
16.	b	17.	c						

3 – 'C' TOKENS

3.1 INTRODUCTION:

The smallest individual unit in a program is called token. The C tokens are classified as Keywords, Identifiers, Constants, Operators, Strings and Special symbols.

3.2 CHARACTER SET:

The characters that can be used to form words, numbers, expressions, data types, constants, variables and keywords, etc., The characters in C are grouped into four categories.

1	Alphabets (Letters)	A,B,C,D . . . . . X,Y,Z a,b,c,d. . . . . x,y,z
2	Digits	0,1,2,3,4,5,6,7,8,9
3	Special characters	~ ` ! @ # % $ ^ & * ( ) + - | ? > < , { } [ ] : ; / \ =
4	White spaces	Blank space, horizontal tab, vertical lab, carriage return, new line, form feed

3.3 KEYWORDS & IDENTIFIERS:

In 'C' every word is classified into either a keyword or an identifier. Keywords have fixed meaning and it cannot be changed and it must be written in lowercase. There are 32 keywords in ANSI C.

Example keywords:

Auto double int struct long extern float if
Else goto default for do while short

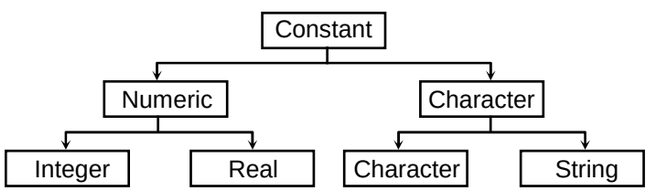
Identifiers are user defined names consisting of a sequence of letters and digits, used to refer the names of the variable, functions, arrays, etc., It must begin with an alphabet or underscore and should not contain white space. Both uppercase and lowercase letters are permitted; an uppercase is not equivalent to the corresponding lowercase letter, although lowercase letters are commonly used. It cannot be a keyword. It can be of any length, most of the C compiles will recognize only first eigth characters.

3.4 CONSTANTS AND VARIABLES:

Constants and variables can be formed by combining the alphabets, numbers and special symbols.

Constants: Constants are fixed values that donot change during the execution of a program. For example in the equation 5X+2Y=45 since 5, 2 and 45 cannot change, these are called Constants, whereas the quantities X & Y can vary or change and hence these are called Variables.

Constant Classification :



Numeric Constants : Numeric constants are classified into 1. Integer constants and 2. Real constants.

1. Integer Constant : An integer constant is a sequence of digits without a decimal point; no commas, no blank spaces are allowed. It could be either positive or negative; if no sign precedes it is assumed to be positive. It ranges from -32768 to 32767. It may be specified in Decimal, Octal or Hexa Decimal notation.

A Decimal Integer constant : It consists of a sequence of one or more decimal digits 0 through 9 preceded by an optional sign, the first digit of the sequence cannot be 0 unless the decimal integer constant is 0.

Ex : 0 276 3412 31467 -7123

An Octal Integer constants : It consists of the digit 0 followed by a sequence of one or more Octal digits. 0 through 7.

Ex : 012 07134 07777

A Hexa Decimal integer constant : It consists of the digit 0, followed by one of the letter x or X, followed by a sequence of one or more Hexadecimal digits 0 through 9 or letter a through f or A through F.

Ex : 0X1F 0XABC 0X9a2F 0XFFFF

2. Real Constant : A real constant is to be a sequence of digits with a decimal point, no commas, blank spaces are allowed. It could be either positive or negative, if no sign precedes it is assumed to be positive. The Scientific notation is often used to express numbers that are very small or very large. Thus 0.000000011 is written as 1.1×10^{-8} and 20000000000 as 2×10^{10}. C provide an exponential form for such numbers that are related to the scientific notation. In exponential form of representation the real value will be represented in two parts. The part appearing before 'e' is called mantissa, where as the part following 'e' is called exponent.

(coefficient)e(Integer) = (coefficient)X10$^{(integer)}$

Character Constants : Character constants are classified into 1. Single character constants and 2. String constants.

1. Single Character Constants : Any character written within single quotes is called ***Character Constant.*** A character constant represents an integer value equal to the numerical value of the character in the machine's character code is known as ASCII value. Since each character constant represents an integer value it is also possible to perform arithmetic operations on character constants.

Example :

'x', 'k', '6', ':', '!', '?', ' ', '*', '+', '"'

2. String constant : A String constant is a sequence of characters enclosed within a pair of double quotes. The characters may be letters, numbers, special characters and blank space and etc., A string constant is also known as a String Literal.

Example :

"hello" "1999" "5+4+6" "Good Bye"

The compiler automatically places a special character NULL at the end of the string constant. The NULL character at the end of the string is known as the delimiter of the string. The difference between a character constant and string constant is : 'p' character constant gives the ASCII value of the character p whereas the string constant "p" contains the character p and NULL.

Variables : A variable is a data name which can be used to store data and a variable may take different values at different times, during execution.

For example in the equation 5X+2Y=45 since 5, 2 and 45 cannot change, these are called Constants, whereas the quantities X & Y can vary or change; hence these are called Variables.

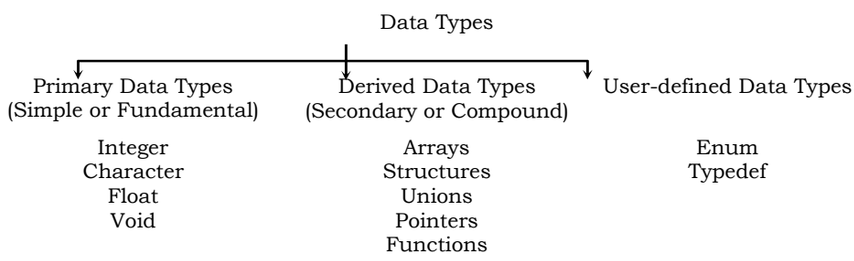
Rules for Constructing Variable Names :

1. A variable name is any combination of 1 to 8 alphabets, digits or underscore. Some compilers allow variables names whose length could be upto 40 characters. Still it would be safe to stick to its rule of 8 characters.
2. The first character in the variable name must be an alphabet.
3. No commas or blank spaces are allowed.
4. No special symbol other than an underscore can be used.

Example: Bas_pay Net_sal month

3.5 DATA TYPES AND SIZES :

Data is represented using numbers or characters. All compilers support a variety of data types. Every programming language has its own data types. Each data type has predetermined memory requirement and an associated range of legal values. C supports four different classes of data types namely (1) Basic or Primary Data types (2) Derived Data types (3) User-defined Data types and (4) Pointer Data types.



Integer Data type : C offers three different integer data types they are : int, short int and long int. The difference between these three integers is the number of bytes to occupy and the range of values.

Type	Bytes Required	Range
Short int	2	–32768 to 32767 (-2^{15} to $2^{15}-1$)
Int	2	–32768 to 32767 (-2^{15} to $2^{15}-1$)
Long int	4	–2147483848 to 2147483847(2^{31} to $2^{31}-1$)
Unsigned short int	2	0 to 65535 (0 to $2^{16}-1$)
Unsigned int	2	0 to 65535 (0 to $2^{16}-1$)
Unsigned long int	4	0 to 4294967295 (0 to $2^{32}-1$)

Float Data type : Like integers floats are divided into three types. They are float, double and long double. The difference between these three floats are the number of bytes to occupy and the range of values.

Type	Description	Size	Range
Float	Single Precession	4	3.4E-38 to 3.4E+38
Double	Double Precession	8	1.7E-308 to 1.7E+308
Long Double	Extended Precession	10	3.4E-4932 to 3.4E+4932

Characters Data type : A char data type can store an element of machine's character set and will occupy 1 Byte. It is of two types, they are signed char and unsigned char. The difference between these two types is the range of values.

Type	Bytes Required	Range
Signed char	1	-128 to +127
Unsigned char	1	0 to 255

3.6 DECLARATION OF VARIABLES :

In any programming language any variable used in the program must be declared before it is used. This declaration tells the compiler what the variable name and what type of data it holds. In C language the declaration of a variable should be done in the declaration part of the program. The type declaration statement is usually written at the beginning of the C program.

Syntax :

<Data_type> <var1>, <var2>, . . ., <varn>;

Example : int i, count;
float price, salary;
char c;

Scope of variables : Scope of variables implies to the availability within a program. Variables have two types of scopes, Local and Global. A variable with a global scope is accessible to all statements in a program but the one with local scope is restricted to be accessed only by certain selected statements in the program, in which it is defined. The global variables are declared outside all functions where as local variables are defined inside a function.

3.7 USER-DEFINED DATA TYPE :

The user defined data types are two types, they are :

1. **Type definition :** The users can define an identifier that represents an existing data type by a feature known as ***"type definition".*** The user defined data type identifier can later be used to declare variables.

General Form :

typedef type identifier;

Here type refers to an existing data type and identifier refers to the new name given to the data type.

Ex : **typedef int** sno;
typedef float salary;

Here ***sno*** symbolizes ***int,*** and ***salary*** symbolizes ***float.*** These can be used to declare variables as follows.

sno c1, c2;
salary e1, e2;

2. **Enumerated data type :** Another user-defined data type is ***enumerated data type.*** The user-defined enumerated data type can be used to declare variables that can have one of the values out of many enumeration constants. After that we can declare variables to be of this new type

General Form :

enum identifier {value1, value2, . . . , valueN};

Here the identifier 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 (known as enumeration constants).

enum identifier v1, v2, . . , vn;

The enumerated variables v1, v2, . . , vn can only have one of the values value1, value2, . . , valueN.

Ex :

enum month {January, February, . . , December};

enum month month_st, month_end;

(or)

enum month {January, February, . . , December} month_st, month_end;

Here the definition and declaration of enumerated variables can be combined in one statement.

3.8 ASSIGNING VALUES :

Values can be assigned to variables by using the assignment operator. An assignment statement implies that the value of the variable on the left of the equal sign is equal to the value of the quantity on the right.

Syntax :

variable_name = constant

'C' permits multiple assignments in one line like

a=10; b=20; c=30;

Variables declared can be assigned or initialized using an assignment operator '='. C permits the initialization of more than one variable in one statement by using multiple assignment operators.

int x = y = 10;

x = y = z = 10;

a = b = c = temp;

3.9 DEFINING SYMBOLIC CONSTANTS :

Some constants may appear repeatedly in a number of places in the program. For example the mathematical constant "**pi**" with the value 3.142. In any case if you want to change the value of pi we have to search throughout the program, if any value is left, the program may produce wrong result. To avoid this disadvantage C provides to define symbolic constants.

General form :

#define symbolic-name value of constant

Example :

#define PI 3.142

#define MAX 100

3.10 OBJECTIVE TYPE QUESTIONS

1. Which is not a character of C? [ ]
   a) $ b) ^ c) ~ d) |
2. Which is not a C keyword? [ ]
   a) const b) main c) sizeof d) void
3. Identify the scalar datatype: [ ]
   a) double b) union c) function d) array
4. The qualifier that may precede float is : [ ]
   a) signed b) unsigned c) long d) none of the above
5. Identify the Octal constant: [ ]
   a) 637 b) 0x25 c) -0756 d) 06.25
6. Identify the invalid constant: [ ]
   a) " " b) ' ' c) 'in' d) '\b'
7. Symbolic constants are defined as: [ ]
   a) #define s1 s2 b) #define s1=s2
   c) #define s1 s2; d) #define s1=s2;
8. Identify the C token(s): [ ]
   a) keywords b) constants c) operators d) all above
9. Statement terminator in 'C' is represented by: [ ]
   a) : b) blank c) ; d) \n

10. Which is an invalid variable name: [ ]
    a) Ints b) Xx c) net-salary d) floating
11. Identify invalid identifier [ ]
    a) NET _$ b) BINGO d) _account d) _4
12. Identify the wrong statement: [ ]
    a) unsigned long int a,b; b) long float f1;
    c) long double ld; d) signed a;
13. A block is enclosed with pair of : [ ]
    a) { } b) ( ) c) [ ] d) < >

14. The declaration of C variables can be done [ ]
    a) anywhere in the program b) in executable part
    c) in declaration part d) at the end of the program
15. A short integer variable occupies memory [ ]
    a) 1 byte b) 2 bytes c) 4 bytes d) 8 bytes
16. A character variable can store only [ ]
    a) 1 character b) 10 characters
    c) 100 characters d) none of the above
17. A variable name can be starts with [ ]
    a) asterisk symbol (*) b) hash symbol (#)
    c) underscore symbol (_) d) none of the above
18. The variables are initialized using [ ]
    a) greater than (>) b) equal to (=)
    c) double equal to (==) d) none of the above
19. C variable can start with [ ]
    a) an alphabet b) a number
    c) dot (.) d) none of the above
20. Identifiers are]
    a) C statements b) user-defined names
    c) reserved words d) none of the above

ANSWERS

1.	a	2.	b	3.	a	4.	d	5.	c
6.	c	7.	a	8.	d	9.	c	10.	c
11.	a	12.	d	13.	a	14.	c	15	b
16.	a	17	c	18.	b	19.	a	20	b

4 – OPERATORS AND EXPRESSIONS

4.1 OPERATORS :

An operator is a symbol which represents a particular operation that can be performed on data. The data itself (which can be either a variable or a constant) is called the 'operand'. Expressions are made by combining operators and operand. C is rich in use of different operators.

1. Arithmetic
2. Assignment
3. Relational
4. Unary
5. Bitwise
6. logical or Boolean
7. Conditional statement.
8. Special

4.1.1 Arithmetic Operators :

The arithmetic operators in 'C' language are '+' (Addition), '-' (Subtraction), '*' (Multiplication), '/' (Division) and '%' (modulus). These operators are called 'binary' operators as they operate on two operands at a time. Each operand can be int, float or char.

Ex:-

```
int x,y,z;
    z = x + y;
    z = x - y;
    z = x * y;
    z = x / y;
    z = x % y;
```

If both operands are integers, then the expression is called an ***integer expression*** and the operation is called ***integer arithmetic.*** Integer arithmetic always yields an integer value. If both operands are real, then the expression is called an ***real expression*** and the operation is called ***real arithmetic.*** A real operand may be either in decimal or exponential notation. Real arithmetic always yields an real value. The modulus (%) operator cannot be used for real operands. If one of the operand is real and the other is integer then the expression is called ***mixed-mode arithmetic*** expression. Here only the real operation is performed and the result is always in real form.

4.1.2 Assignment Operators :

Assignment operators are used to assign the result of an expression to a variable, Usual assignment operator is '='. In addition C has a set of ***'shorthand'*** assignment operators.

Syntax :

```
v op= exp
```

Here v is a variable, exp is an expression and op is a C binary arithmetic operator. The operator op= is known as the shorthand assignment operator. The shorthand assignment operators are '+=' is called add assignment operator, '-=' is called minus assignment operator, '*=' is called multiply assignment operator, '/=' is called divide assignment operator and '%=' is called modulus assignment operator.

Ex :

```
x += y is equivalent to x = x + y
x -= y is equivalent to x = x – y
x *= y is equivalent to x = x * y
x /= y is equivalent to x = x / y
x %= y is equivalent to x = x % y
```

4.1.3 Relational operators:

The relational and equality operators are used to test or compare the values between two operands. The relational and equality operators produce an integer result to express the condition of the comparison. If the condition is false then the integer result is 0 (zero) otherwise the integer result is non zero.

<	less than	!=	not equal to
>=	less than or equal to	==	equal to
>	greater than	=	assignment
>=	greater than or equal to.		

4.1.4 Unary Operator :

'C' includes a class of operator that acts upon a single operand to produce a new value, such operators are known as Unary operators. Unary operators usually preceded their single operand. Most commonly used unary operators are 1) Unary minus operator and 2) Increment and Decrement operators

1. **Unary minus :** Where a minus sign precedes a numerical constants, variables or an expressions. Where as the unary minus operator is different from the arithmetic minus operator. Thus a negative number is actually an expression consisting of unary minus operator.

   Ex : x = -y;

2. **Increment and Decrement operators:** The increment (++) and decrement (--) operators are add one and subtract one. These operators operate on only one operand and the operand has to be a variable. The add one or subtract one of the value either before or after the value of the variable is used. If the operator appears before the variable, it is called a *prefix* operator. If the operator appears after the variable, it is called a *postfix* operator.

   a++ and ++a is the same when the statements are independent. For example if a = 5 then a++ and ++a are same the value will be 6.

   When the prefix ++ (or --) is used in an expression, the variable is incremented (or decremented) first and then the expression is evaluated using the new value of the variable. Where as the postfix ++ (or --) is used in an expression, the expression is evaluated first using with the original values of the variables and then the variable is incremented (or decremented) by one.

   Consider the following :

   a = 5;

   b = a++;

   In this case the value of a would be 6 and b would be 5. If we write the above statement as

   a = 5;

   b = ++a;

   then the value of a would be 6 and b would be 6.

4.1.5 Bitwise operators :

The smallest element in the memory on which we are able to operate is a bit. C supports several Bitwise operators. These permit the programmer to access and manipulate individual bits within a piece of data. The 'C' Bitwise operators can operate on ints and chars but not on float.

&	Bitwise AND
|	Bitwise OR
>>	right shift
<<	left shift
~	one's complement
^	Bitwise XOR (exclusive OR)

1. Bit wise and operator : The operator is represented as '&' and operates on two operands. While operating upon these two operands they are compared on a bit-by-bit basis. (Both the operands must be of same type either chars or ints).

The truth table for & is:

&	0	1
0	0	0
1	0	1

```
Ex :        X  =  0000 0111   ( = 7)
            Y  =  0000 1000   ( = 8)
Z = X & Y   =  0000 0000   ( = 0)
```

2. Bit wise or operator : The operator is represented as '|' and operates on two operands. While operating upon these two operands they are compared on a bit-by-bit basis. (Both the operands must be of same type either chars or ints).

The truth table for | is :

|	0	1
0	0	1
1	1	1

```
Ex :        X  =  0000 0111   ( = 7)
            Y  =  0000 1000   ( = 8)
Z = X | Y   =  0000 1111   ( = 15)
```

3. **One's complement :** The operator is represented as '~' and operates on one operand. For a binary number if we take one's complement all zero's become 1 and one's become 0's

Ex: X = 0001

~X = 1110

4. **Right Shift :** This operation will divide the given number by 2^s i.e. $x = n/2^s$, where n is the input nuber and s is the number of shifts.

Example :

x = 16

x = x>>3 After execution x will be 2.

5. **Left Shift :** This operation multiplies the given number by 2^s i.e. $x = n * 2^s$, where n is the input nuber and s is the number of shifts.

Example :

x = 4

x = x<<2 After execution x will be 16.

4.1.6 Logical operators:

Logical operators are used to combine two or more relations. The logical operators are called Boolean operators. Because the tests between values are reduced to either true or false, with zero being false and one being true.

!	not operation	(logical not)
&&	and operation	(logical and)
||	or operation	(logical or)

The expressions can be connected like the following

(expression1) &&/|| (expression2)

Operands		Results	
Exp1	**Exp2**	**Exp1&&exp2**	**Exp1||exp2**
0	0	0	0
0	Non zero	0	1
Non zero	0	0	1
Non zero	Non zero	1	1

4.1.7 Conditional operator :

The conditional operator ? and : are sometimes called ternary operator since it operates on three operands, and it is a condensed form of an if-then- else 'C' statement. The general form is:

exp1 ? exp2 : exp3

Ex: y = (x > 5 ? 3 : 4) is equivalent to

```
if (x>5)
then
    y = 3;
else
    y = 4;
```

4.1.8 Special operators:

Some commonly used special operators are.

1. **Comma operator :** The comma(,) operator permits two different expressions, appears in a situation where only one expression would ordinarily be used. The expressions are separated by comma operator.

   Ex: c = (a=10, b=20, a+b);

   Here first the value 10 is assigned to a, followed by 20 is assigned to b and then calculate a+b and result is assigned to c.

2. **Size of operator :** The size of operator returns the number of bytes the operand occupies in memory. The operand may be a variable, a constant or a data type.

   Ex : sizeof(int) is going to return 2

3. **Address of operator :** The address of operator (&) returns the address of the variable. The operand may be a variable, a constant.

   Ex : m = &n;

Here address of n is assigned to m. This m is not a ordinary variable, it is a variable which holds the address of the other variable.

4. **value at address operator :** The value at address operator (*) returns the value stored at a particular address. The ***'value at address'*** operator is also called ***'indirection'*** operator.

Ex : x = *m;

Here the value at address of m is assigned to x, and m is going to hold the address.

4.2 PRECEDENCE OF OPERATORS :

All arithmetic expressions may contain two or more operators, then we may have some problem about how to get it exactly executed. To answer these questions one has to understand the precedence of operators. The order of priority in which the operations are performed in an expression is called precedence. The precedence of all operators is shown below.

Description	Operator	Rank	Associativity
Function expression	( )	1	Left to Right
Array expression	[ ]		
Unary plus	+	2	Right to Left
Unary minus	-		
Increment/Decrement	++/--		
Logical negation	!		
One's complement	~		
Pointer reference	*		
Address of	&		
Size of an object	Sizeof		
Type cast (conversion)	(type)		
Multiplication	*	3	Left to Right
Division	/		
Modulus	%		
Addition	+	4	Left to Right
Subtraction	-		

Description	Operator	Rank	Associativity
Left shift	<<	5	Left to Right
Right shift	>>		
Less than	<	6	Left to Right
Less than or equal to	<=		
Greater than	>		
Greater than or equal to	>=		
Equality	==	7	Left to Right
Not equal to	!=		
Bitwise AND	&	8	Left to Right
Bitwise XOR	^	9	Left to Right
Bitwise OR	|	10	Left to Right
Logical AND	&&	11	Left to Right
Logical OR	||	12	Left to Right
Conditional	? :	13	Right to Left
Assignment	=	14	Right to Left
	*= /= %=		
	+= -= &=		
	^= |= |		
	<<= >>=		
Comma operator	,	15	Left to Right

4.3 EXPRESSIONS :

An expression is a combination of variables, constants and operators arranged as per the syntax of the language.

Ex :

Algebraic Expression	C Expression
(a – b)(c + d)e	(a – b) * (c + d) * e
$4x^2+8y+10$	4 * x * x + 8 * y + 10
$\left(\frac{a}{b}\right)+c$	(a/b) + c
$\left(\frac{ab}{c}\right)+d$	((a * b) / c) + d

4.4 EVALUATIONS OF EXPRESSIONS :

Expressions are evaluated by using an assignment statement. The expressions is evaluated first and then the result replaces the previous value of the variable on the left hand side. All variables used in the expression must be assigned with values before evaluation is attempted.

Example of evaluation statement are :

```
x = b / c * a;
y = a – b / c + d;
z = a + b – c;
```

Rules for Expression Evaluation :

- First parenthesized sub expressions from left to right are evaluated.
- If parentheses are nested, the evaluation begins with the innermost sub-expression i.e. the innermost parenthesis will be solved first, followed by the second and so on.
- When parentheses are used, the expressions within parentheses assume highest priority.
- The precedence rule is applied for evaluating sub-expressions.
- The associativity rule is applied when two or more operators of the same precedence level appear in a sub-expression. i.e. When two operators of the same priority are found in the expression, precedence is given to the extreme left operator.

4.5 TYPE CONVERSION :

When an operator has different data types of operands it is in ***Mixed mode.*** C supports the use of mixed mode operations in arithmetic expressions. Normally, before performing any operation we have to see that both operands must have the same data type. C conversion one or both the operands to the appropriate data types is called ***Type Conversion.***

Type Conversion can be achieved by two methods, they are Implicit type Conversion and Explicit Type Conversion.

Implicit Type Conversion : In this case one data type is automatically converted into another data type as per the rules described. There are two types of implicit type conversion viz., automatic type conversion and assignment type conversion.

a) **Automatic Type Conversion :** In automatic type conversion, the data type with lower rank is converted automatically into higher rank data before the operation proceeds.

b) **Assignment Type Conversion :** If the two operands in an assignment operation are of different data types.

4.6 OBJECTIVE TYPE QUESTIONS

1. What is the result of 5/4 [ ]
   a) 0 b) 1 c) 5 d) None
2. What is the result of 5%4 [ ]
   a) 0 b) 1 c) 5 d) None
3. What is the result of 32>>3 [ ]
   a) 4 b) 9 c) 0 d) None
4. What is the result of 3<<3 [ ]
   a) 0 b) 9 c) 24 d) None

5. Multiplication of two numbers is performed using [ ]
   a) unary operator b) logical operator
   c) arithmetic operator d) assignment operator
6. The '&' operator displays [ ]
   a) Value of the variable b) Address of the variable
   c) Both a and b d) None of the above
7. What is the value of !1? [ ]
   a) 0 b) 1 c) -1 d) None
8. Number of binary arithmetic operators in C is: [ ]
   a) 5 b) 4 c) 6 d) 7
9. Which is not a valid expression? [ ]
   a) +0XAB5 b) -0525 c) 15- d) +a
10. Which is not a valid expression? [ ]
    a) –p++ b) ++p— c) ++6 d) ++x++
11. Which is not a valid expression? [ ]
    a) ++(a+b) b) y— c) –x— d) ++p+q
12. The equality operator is represent by: [ ]
    a) := b) .EQ. c) = d) ==
13. Identify the logical operator: [ ]
    a) ! b) != c) ~ d) ==
14. Identify the relational operator: [ ]
    a) && b) > c) || d) !
15. The symbol for exclusive OR operator is: [ ]
    a) ^ b) ~ c) & d) |
16. The symbol for one's complement operator is: [ ]
    a) & b) ^ c) ~ d) |
17. The symbol for bitwise AND operator is: [ ]
    a) << b) &= c) && d) &
18. The symbol for left shift operator is: [ ]
    a) < b) << c) <= d) <<<
19. Structural programming approach makes use of: [ ]
    a) modules b) control structures
    c) used defined data types d) all above
20. A statement is differentiated from an expression [ ]
    a) by terminating it by a semicolon
    b) by terminating it by a newline character
    c) by terminating it by a NULL
    d) by terminating it by a blank space

ANSWERS

1.	b	2.	b	3.	a	4.	c	5.	c
6.	b	7.	a	8.	a	9.	c	10.	c
11.	a	12.	d	13.	a	14.	b	15	a
16.	c	17	d	18.	b	19.	d	20	a

5 – INPUT AND OUTPUT

5.1 INTRODUCTION :

'C' language does not have any built-in input/output statements. All input/output operations can be performed through function calls such as printf and scanf. These functions are known as the standard I/O library. A program using these functions must include the standard input-output header file <stdio.h> line :

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

5.2 TYPES OF I/O :

The input/output functions are classified into three categories.

1. **Console I/O functions:** Functions to receive input from keyboard and write output to VDU
2. **Disk I/O functions :** Functions to perform I/O operations on a floppy or Hard Disk.
3. **Port I/O functions :** Functions to perform I/O operations on various ports (serial and parallel).

An input/output functions can be accessed from any where within a program by simply writing the function name followed by a list of arguments enclosed in parentheses.

Console I/O Functions : Console I/O functions are mainly classified into two categories

a) Unformatted console I/O functions

b) Formatted console I/O functions

Unformatted Console I/O functions : Functions that accept a single argument (the argument must be data item representing a string or a character) are concerned with unformatted I/O.

		Char type	String type
Input	:	getch(), getche(), getchar()	gets()
Output	:	putch(), putchar()	puts()

getch() & getche() : These functions will read a single character the instant it is typed without waiting for the enter key to be hit.

Program 5.1 :

```
/* PROGRAM TO DEMONSTRATE getch() AND getche() FUNCTIONS */
#include<stdio.h>
main()
{
        char ch;
        printf(" Hit any key ! : ");
        getch();     /*  character input without echo  */
        printf(" Hit any key ! : ");
        getche();
        /*  character will be echoed on the screen   */
}
```

Explanation : The above program will explain the difference between getch() and getche() functions. The main difference is getche function will echo a character but it is not in getch function.

getchar() : It is very similar to the getche() and getch() functions echoing the character you type on the screen, but requires enter key to hit following the character you typed.

putchar() and putch() : These functions are used to write a character one at a time to the terminal.

Program 5.2 :

```
/*  PROGRAM TO DEMONSTRATE getchar() AND putch() FUNCTIONS  */
#include<stdio.h>
main()
{
        char ch;
        printf(" Hit any key ! : ");
        ch=getchar();
        putch(ch);
}
```

Explanation: In the above program character ch will be read through getchar() function and character will print through putchar() function.

gets() & puts() : gets() receives a string which is an array of characters from the keyboard, puts() function works exactly opposite to gets() function i.e. prints the string on console.

Program 5.3 :

```
/*  PROGRAM TO DEMONSTRATE gets() AND puts() FUNCTIONS  */
#include<stdio.h>
main()
 {

        char name[100];
        puts("Enter a string : ");
        gets(name);
        puts(name);
 }
```

OUTPUT :

```
Enter a string :
I am a Software Engineer
I am a Software Engineer
```

Explanation : In the above program a string will read through the gets function and will ptint it by puts function.

Formatted Console I/O : Functions that accept strings as well as variable number of arguments to be displayed and read in a specified format is formatted I/O.

Type	:	char, int, float, string
Input	:	scanf()
Output	:	printf()

The above two functions are used to supply input from keyboard in a fixed format and obtain output in a specified format on the screen.

printf function : Output data can be written from the computer on to a standard output device using this library function. This function can be used to output any combination of numerical values, single character and strings.

The general format is :

printf("control string", arg1, arg2, arg3...);

where the control string refers to a string contains formatting information. When printf() is called it opens the format string and reads a character at a time. If the character reads % or \ it does not print it but reads the next character, have a special meaning to printf.

format descriptors :

%d	for int and short int
%ld	for long int
%u	for unsigned int and unsigned short int
%lu	for long unsigned int
%f	for float
%lf	for double
%Lf	for long double
%c	for char
%s	for string.
%o	for octal

%x for hexa decimal

Escape Sequences :

\n	new line
\t	horizontal tab
\v	vertical tabulator
\a	to beep the speaker
\'	single quote
\"	double quotes
\?	question mark
\\	back slash
\0	NULL.

Output of Integer Number : The format specification for printing an integer numbers is %wd. Here w specifies the minimum field width for the output. If a number is greater than the specified field width the number will be printed fully. d specifies that the value to be printed, is an integer. The following example program illustrates the output of an integer in different formats

Program 5.4 :

```
/*  OUTPUT OF INTEGER NUMBERS UNDER VARIOUS FORMATS   */
#include <stdio.h>
main()
 {
      int i=3214;
      clrscr();
      printf(" i = %d\n",i);
      printf(" i (%%3d) = %3d\n",i);
      printf(" i (%%7d) = %7d\n",i);
      printf(" i (%%-7d) = %-7d\n",i);
      printf(" i (%%010d) = %010d\n",i);
      printf(" i (%%.10d) =  %.10d\n",i);
      printf(" i (%%o) = %o\n",i);
      printf(" i (%%x) = %x\n",i);
      printf(" i (%%#o) = %#o\n",i);
      printf(" i (%%#x) = %#x\n",i);
      printf(" i (%%6d) = %6d\n",-i);
 }
```

OUTPUT :

i	=	3	2	1	4					
i (%3d)	=	3	2	1	4					
i (%7d)	=				3	2	1	4		
i (%-7d)	=	3	2	1	4					
i (%010d)	=	0	0	0	0	0	0	3	2	1	4
i (%.10d)	=	0	0	0	0	0	0	3	2	1	4
i (%o)	=	6	2	1	6					
i (%x)	=	c	8	e						
i (%#o)	=	0	6	2	1	6				
i (%#x)	=	0	X	c	8	e				
i (%6d)	=		-	3	2	1	4			

Explanation : In the above program integer i is initialized with 3214, it is printed with various formats.

Output of Real Numbers : The real numbers may be displayed in decimal notation using the format specification of %w.p f. Here w indicates the minimum number of positions that are to be used for display the value and p indicates the number of

digits to be displayed after the decimal point (precision). The value is rounded to p decimal places and printed. The default precision is 6 decimal places.

We can also display a real number in exponential notation by using the specification %w.p e. The following Example program illustrates the output of a real number in different formats;

Program 5.5 :

```
/*  OUTPUT OF REAL NUMBERS IN VARIOUS FORMATS   */
#include <stdio.h>
main()
 {
      float i=2500.321;
      clrscr();
      printf(" i (%%f) =  %f\n",i);
      printf(" i (%%f) =  %f\n",-i);
      printf(" i (%%+.0f) =  %+.0f\n",i);
      printf(" i (%%-.0f) =  %-.0f\n",i);
      printf(" i (%%8.2f) =  %8.2f\n",i);
      printf(" i (%%6.8f) =  %6.8f\n",i);
      printf(" i (%%2.2f) =  %2.2f\n",i);
      printf(" i (%%10.2e) =  %10.2e\n",i);
      printf(" i (%%09.2f) =  %09.2f\n",i);
      printf(" i (%%9.2f) =  %9.2f\n",i);
      printf(" i (%%012.2f) =  %012.2f\n",i);
      printf(" i (%%12.2f) =  %12.2f\n",i);
      printf(" i (%%8.2f) =  %8.2f\n",i);
      printf(" i (%%#10.0f) =  %#10.0f\n",i);
      printf(" i (%%e) =  %e\n",i);
      printf(" i (%%*.*f 8 2) =  %*.*f",8,2,i);
 }
```

OUTPUT :

i (%f)	=	2500.321045
i (%f)	=	-2500.321045
i (%+.0f)	=	+2500
i (%-.0f)	=	2500
i (%8.2f)	=	2500.32
i (%6.8f)	=	2500.32104492
i (%2.2f)	=	2500.32
i (%10.2e)	=	2.50e+03
i (%09.2f)	=	002500.32
i (%9.2f)	=	2500.32
i (%012.2f)	=	000002500.32
i (%012.2f)	=	2500.32
i (%8.2f)	=	2500.32
i (%#10.0f)	=	2500.
i (%e)	=	2500321e+03

i (%*.*f 8 2) = | | 2 | 5 | 0 | 0 | . | 3 | 2 |

Explanation : In the above program float i is initialized with 2500.321, it is printed with various formats.

Output of Characters and Strings : The Characters and Strings may be displayed by using the format specification of %w.p f. Here w specifies the field for display and p instructs that the first p characters of the string are to be displayed. The display is right-justified. The following Example program illustrates the output of Characters and Strings in different formats;

Program 5.6 :

```
/*  PRINTING OF CHARACTERS & STRINGS IN VARIOUS FORMATS   */
#include<stdio.h>
main()

 {
      char name[50]="SHREETECH Computers",ch='S';
      clrscr();
      printf(" ch = %c\n",ch);
      printf(" ch = %3c\n",ch);
      printf(" ch = %6c\n",ch);
      printf(" name = %s\n",name);
      printf(" name = %15s\n",name);
      printf(" name = %*s\n",2,name);
      printf(" name = %20.10s\n",name);
      printf(" name = %-20.10s\n",name);
      printf(" name = %.5s\n",name);
 }
```

OUTPUT:

Ch = | S |

Ch = | | | S |

Ch = | | | | | | S |

name = | S | H | R | E | E | T | E | C | H | | C | o | m | p | u | t | e | r | s |

name = | S | H | R | E | E | T | E | C | H | | C | o | m | p | u | t | e | r | s |

name = | S | H | R | E | E | T | E | C | H | | C | o | m | p | u | t | e | r | s |

name = | | | | | | | | | | | S | H | R | E | E | T | E | C | H |

name = | S | H | R | E | E | T | E | C | H | | | | | | | | | | |

name = | S | H | R | E | E |

Explanation : In the above program character ch is initialized with 'S' and string name is initialized with "SREETECH Computers", it is printed with various formats.

scanf Function: This function can be used to read any combination of numerical values, single character and string.

The general format is :

scanf("control string", arg1, arg2, arg3...)

Where the control string refers to a string certain required formatting information, arg1, arg2, ..., argn are arguments that represent the individual data items.

Ex:- int n;
scanf("%d",&n);

Here the scanf() function gets the value from user and store the value in the address(memory location) of the variable 'n'. Which is represented as &n. Hence, when we use '&' symbol we refers to the address of the variable.

scanf() has got its own limitations. Though both scanf() & gets() are used for inputting a string, scanf() will not allow to input a string with blank spaces.

5.3 BLOCKS :

A group of declarations and statements surrounded by the opening and closing braces ({ }) is called ***a Compound statement*** or a ***Block.*** A block may hold variables, statements, structures, expressions, control structures, etc.,

which are necessary for a program. Any variables declared inside a block are not available outside of block, declaration should be done at the beginning of the block. The variables declared inside any block are called block variables. A block is syntactically equivalent to a single statement no semicolon is placed after the closing brace that ends a block. A program can hold any no. of blocks, also it is possible to insert a block within another block. General format of blocks :

```
{           // initiation of a block
   variables;
   expressions;
}           // termination of a block


{           // initiation of block1
   ------------
   {        // initiation of block2
      -----------
   }        // termination of block2
   -------------
}           // termination of block 1
```

Ex :

```
{
    int x, y=5;
    x=y*y*y;
    printf("%d \t %d \n", y, x);
}
```

5.4 OBJECTIVE TYPE QUESTIONS

1. To scan a (float) and b (double), which scanf() statement do you use? [ ]
   a) scanf("%f%f",&a,&b); b) scanf("%Lf%Lf",&a,&b)
   c) scanf("%f%Lf",&a,&b); d) scanf("%f%lf",&a,&b);
2. To printout a (float a=5.43) and b (double b=2.14), which printf() statement do you use? [ ]
   a) printf("%f%f",a,b); b) printf("%Lf%f",a,b)
   c) printf("%f%Lf",a,b); d) printf("%Lf%Lf",a,b);
3. What is the output of the following code [ ]

```
main()
{
    printf("\n%%%%");
}
```

   a) % b) %% c) %%% d) %%%%

ANSWERS

1.	d	2.	a	3.	b				

5.5 EXAMPLE PROGRAMS

Program 5.7 :

```
/*   PROGRAM TO DEMONSTRATE sizeof OPERATOR     */
#include<stdio.h>
main()

 {
       short int a;
       int b;
       long int c;
       float d;
       double e;
       long double f;
       char g;
       printf("short int a\t int b\t long int c\t float d\n");
       printf("double e\t long double f\t char g\n\n");
       printf("Bytes occupied by a is : %d\n",sizeof(a));
       printf("Bytes occupied by b is : %d\n",sizeof(b));
       printf("Bytes occupied by c is : %d\n",sizeof(c));
       printf("Bytes occupied by d is : %d\n",sizeof(d));
       printf("Bytes occupied by e is : %d\n",sizeof(e));
       printf("Bytes occupied by f is : %d\n",sizeof(f));
       printf("Bytes occupied by g is : %d\n",sizeof(g));
       printf("short int occupies %d bytes \n",sizeof(short int));
       printf("int occupies %d bytes \n",sizeof(int));
       printf("long int occupies %d bytes \n",sizeof(long int));
       printf("float occupies %d bytes \n",sizeof(float));
       printf("double occupies %d bytes \n",sizeof(double));
       printf("long double occupies %d bytes \n",sizeof(long double));
       printf("char occupies %d bytes \n",sizeof(char));
       printf("Bytes occupied by '5' is : %d\n",sizeof('5'));
       printf("Bytes occupied by 5 is : %d\n",sizeof(5));
       printf("Bytes occupied by 5.0 is : %d\n",sizeof(5.0));
 }
```

OUTPUT :

```
short int a      int b   long int c      float d
double e         long    double f        char g
Bytes occupied by a is : 2
Bytes occupied by b is : 2
Bytes occupied by c is : 4
Bytes occupied by d is : 4
Bytes occupied by e is : 8
Bytes occupied by f is : 10
Bytes occupied by g is : 1
short int occupies 2 bytes
int occupies 2 bytes
long int occupies 4 bytes
float occupies 4 bytes
double occupies 8 bytes
long double occupies 10 bytes
char occupies 1 bytes
Bytes occupied by 'c' is : 2
Bytes occupied by 5 is : 2
Bytes occupied by 5.0 is : 8
```

Explanation : The above program will specify the number of bytes occupied for each data type, the numbers 5, 5.0 and character 'c'.

Program 5.8 :

```
/*   PROGRAM TO PRINT PREFIX AND SUFIX CHARACTERS     */
#include<stdio.h>
main()
```

```
{
    char c;
    clrscr();
    printf("\nEnter any single character  : ");
    c=getchar();
    printf("\nThe character prefix to character %c is  : ",c);
    putchar(c-1);
    printf("\nThe character suffix to character %c is   : ",c);
    putchar(c+1);
}
```

OUTPUT :

Enter any single character : e
the character prefix to character e is : d
the character suffix to character e is : f

Explanation : The above accepts a character through getchar function as c, it is printing its prefix and suffix characters. Here c is incremented and decremented by 1, it is showing that chacters also performing addition and subtractions.

Program 5.9 :

```
/*  PROGRAM TO ACCEPT THREE SIDES OF TRIANGLE AND CALCULATE THE
AREA AND CIRCUMFERENCE OF IT  */
#include<stdio.h>
#include<math.h>
main()
{
    float a,b,c,cir;
    float s,area;
    printf("Enter first side of a triangle  :   ");
    scanf("%f",&a);
    printf("Enter second side of a triangle  :   ");
    scanf("%f",&b);
    printf("Enter third side of a triangle  :   ");
    scanf("%f",&c);
    cir=a+b+c;
    s=(a+b+c)/2;
    area=sqrt((double)(s*(s-a)*(s-b)*(s-c)));
    printf(" \nThe area of the given triangle is  :  %f\n",area);
    printf(" \nThe circumference of the given triangle is :  %f\n",cir);
    getchar();
}
```

OUTPUT:

Enter first side of a triangle : 4
Enter second side of a triangle : 5
Enter third side of a triangle : 6
The area of the given triangle is : 9.921567
The circumference of the given triangle is : 15.000000

Explanation : The above accepts three float values from the user as sides of triangle. Area and circumference will be calculated by the formula.

Program 5.10 :

```
/*   PROGRAM TO ACCEPT TWO VARIABLES AND INTERCHANGE IT BY WITHOUT
USING TEMPORARY VARIABLE   */
#include<stdio.h>
main()

{
    int a,b,t;
    printf("Enter the first number : ");
```

```
        scanf("%d",&a);
        printf("Enter the second number : ");
        scanf("%d",&b);
        printf("Before interchanging A = %d\t B = %d\n",a,b);
        a=a+b;
        b=a-b;
        a=a-b;
        printf("After interchanging A = %d\t B = %d\n",a,b);
}
```

OUTPUT:

```
Enter the first number : 13
Enter the second number : 51
Before interchanging A = 13    B = 51
After interchanging A = 51     B = 13
```

Explanation : The above program accepts two integers as a and b. These values are interchanged by formula and printed.

6 – CONTROL STATEMENTS

6.1 INTRODUCTION :

In any program statements are normally executed. We have number of situations repeat a group of statements until certain specified condition or change the order of execution of statements. C supports two types control statements, they are

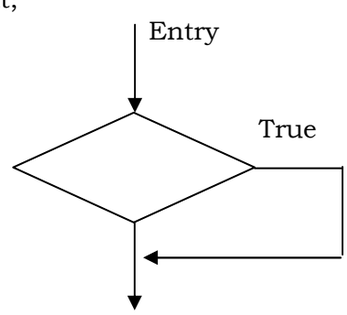
1. Non-iterative Statements : The following are the non-iterative statements
   a) Simple if statement
   b) if-else statement
   c) nested if-else statement
   d) else-if ladder statement
   e) ternary operator
   f) switch statement
2. Iterative Statements : The Iterative statements allow a set of instructions to be performed until a certain condition is reached. C provides three different types of loops namely.
   a) While loop
   b) Do-While loop
   c) For loop

6.2 SIMPLE IF STATEMENT:

The if statement is used to specify conditional execution of program statements or a group of statements enclosed in braces.

The general format is :

```
if (condition)
    statement;
```



Ex:-

Program 6.1 :

```
/*  PROGRAM TO PRINT ABSOLUTE VALUE OF THE GIVEN INTEGER  */
#include<stdio.h>
main()

 {
      int x;
      printf(" Enter any integer number  : ");
      scanf("%d",&x);
      if(x<0)
            x=-x;
      printf("Absolute value of the given number is :  %d\n",x);
 }
```

OUTPUT:

```
Enter any integer number  : -10
Absolute value of the given number is :  10
```

Explanation : This program accepts an integer number as x from user. The entered number is checked by if statement. If it is less than zero it will be multiplied by -1. Finally it prints the absolute value of the given integer.

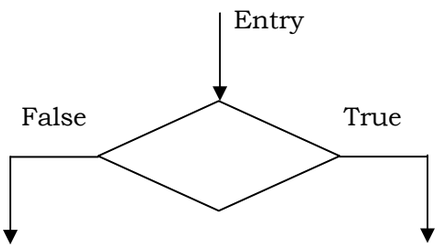
6.3 IF - ELSE STATEMENTS :

There are situations when there are two groups of statements and it is desired that one of them be executed if some condition is true and the other be executed if the condition is false. In such situations we make use of if - else statement.

The general format :

```
if (condition)
```

```
        statement 1;
    else
        statement 2;
```



Ex:-
Program 6.2 :

```
/*  PROGRAM TO FINDOUT THE ACCEPTED NUMBER IS POSITIVE OR NEGATIVE  */
#include<stdio.h>
main()

 {
    int x;
    printf(" Enter any integer number  : ");
    scanf("%d",&x);
    if(x<0)
        printf(" The given number is negative \n");
    else
        printf(" The given number is positive \n");
 }
```

OUTPUT:

```
Enter any integer number :  -12
The given number is negative
```

Explanation : This program accepts an integer number as x from user. The entered number is checked by if statement. If it is less than zero, it will be print "The given number is negative" otherwise it will prints "The given number is positive".

6.4 NESTED IF - ELSE STATEMENTS :

When a series of decisions are involved, we may have to use more than one if-else statements in nested form as follows:

The general format :

```
If (Condition 1)
{
    if (Condition 2)
    {
        Statement 1;
    }
    else
    {
        Statements 2;
    }
}
else
{
    if (Condition 3)
    {
        Statements 3;
    }
    else
    {
        Statements 4;
    }
```

```
        }
        Statement 5;
```

Ex :

Program 6.3 :

```
/*  PROGRAM TO FINDOUT THE HIGHEST NUMBER FROM THE GIVEN THREE NUMBERS  */
#include<stdio.h>
main()

{
    int x,y,z,max;
    printf(" Enter first number  : ");
    scanf("%d",&x);
    printf(" Enter second number : ");
    scanf("%d",&y);
    printf(" Enter third number : ");
    scanf("%d",&z);
    if(x<y)
    {
        if(y<z)
            max=z;
        else
            max=y;
    }
    else
    {
        if(x<z)
            max=z;
        else
            max=x;
    }
    printf(" max number of %d %d %d is : %d\n",x,y,z,max);
}
```

OUTPUT:

```
Enter first number : 8
Enter second number : 23
Enter third number : 9
Max number of 8 23 9 is : 23
```

Explanation : This program accepts three integer numbers as x, y and z from user. The entered numbers are checked one another and the highest number will be saved in max variable. Finally, it wil print the max value i.e. max value of the given three numbers.

6.5 ELSE - IF LADDER :

Here the conditions are evaluated from the top (of the ladder) to downwards. As soon as a true condition is found, the statements associated with it is executed and the rest of the ladder is bypassed. The last else is handles the default case.

The general format :

```
    if (condition)
        statement 1;
    else if (condition)
                statement 2;
          else if (condition)
                      statement 3;
                else
                      statement 4;
```

Ex:

Program 6.4 :

```
/* PROGRAM TO IDENTIFYING WHICH CHARACTER IS ENTERED  */
#include<stdio.h>
main()

{
```

```
    int c;
    printf("Enter any character : ");
    c=getchar();
    if (c>='a' && c<='z')
        printf("The given character is LOWERCASE character\n");
    else if (c>='A' && c<='Z')
            printf("The given character is UPPERCASE character\n");
     else if (c>='0' && c<='9')
                printf("The given character is DIGIT\n");
       else
                printf("The given character is SPECIAL character\n");
}
```

OUTPUT:

Enter any character : 7

The given character is DIGIT

Explanation : The above program to accept a character from the input and identify the character type by using else – if ladder. Each if statement is checking two conditions combining with && operator.

6.6 TERNARY OPERATOR :

C provides condition evaluation operator called the ternary operator in the form of the ? symbol.

The general format :

(condition) ? exp1 : exp2

The ? operator evaluates the condition that precedes it. it is true, it returns exp1 and returns exp2 otherwise.

Ex:-

```
n=n>0?n+10:-n;      (OR)      if(n>0)
                                  n+=10;
                              else
                                  n=-n;
```

Program 6.5 :

```
/*  PROGRAM TO FIND OUT THE MAXIMUM NUMBER FROM THE GIVEN TWO NUMBERS BY USING TERNARY OPERATOR  */
#include<stdio.h>
main()

 {
    int x,y,max;
    printf(" Enter first number  : ");
    scanf("%d",&x);
    printf(" Enter second number : ");
    scanf("%d",&y);
    max=x>y?x:y;
    printf(" max number of %d %d is : %d\n",x,y,max);
 }
```

OUTPUT:

Enter first number : 43

Enter second number : 12

Max number of 43 12 is : 43

Explanation : The above program to accepts two integers as x & y. Both variables checked by ternery operator and highest value will be stored max variable.

6.7 SWITCH STATEMENT :

'switch' statement works in the same way as 'if-else-if', but it is more elegant. The SWITCH statement is a special multi-way decision maker that tests whether an expression matches one of a number of constancy values, and branches accordingly. Switch differs from if-else-if because switch can test for only equality, whether if can evaluate logical expression. The 'switch' statement is often used to process key board commands like menu options.

The general format :

```
switch(expression)
```

```
{
    case var1:
            statement 1;
            break;
    case var2:
            statement 2;
            break;
    case var3:
            statement 3;
            break;
            .
     default:
            statement n;
            break;
}
```

Note that in the above structure switch, case, break and default are C keywords.

Ex :

Program 6.6 :

```
/*  PROGRAM TO DEMO ON SWITCH STATEMENT   */
#include<stdio.h>
main()

{
    int x,y,rst;
    char opt;
    printf(" Enter first number  : ");
    scanf("%d",&x);
    printf(" Enter second number : ");
    scanf("%d",&y);
    getchar();
    printf(" Enter your option +  -  X  /  %%  : ");
    scanf("%c",&opt);
    switch(opt)
    {
        case '+':
            printf(" %d  +  %d  =  %d\n",x,y,x+y);
            break;
        case '-':
            printf(" %d  -  %d  =  %d\n",x,y,x-y);
            break;
        case 'X':
            printf(" %d  X  %d  =  %d\n",x,y,x*y);
            break;
        case '/':
            printf(" %d  /  %d  =  %d\n",x,y,x/y);
            break;
        case '%':
            printf(" %d  %%  %d  = %d\n",x,y,x%y);
            break;
        default:
            printf("No operation\n");
            break;
    }
}
```

OUTPUT:

```
Enter first number : 12
Enter second number : 8
Enter your option +  -  X  /  %  : +
12  +  8  =  20
```

Explanation : The above program to accepts two integers as x & y along with an arithmetic operation symbol. Based on the operator it perfers operation, the result will be displayed.

6.8 WHILE LOOP :

The while loop in the C starts with while keyword, followed by a parenthesized boolean condition, and has a set of statements which constitute the body of the loop.

The general format :

```
while(expression)
 {
        statement 1;
        statement 2;
        statement 3;
                .
        statement n;
 }
```

As soon as execution reaches the while loop, the specified condition is tested. If it is found to be true, it will enter into the body of the loop. Once it reaches the closing brace of the body automatically loop back to the top and test the condition freshly now, and if it is true re-enter the body and so on, till the controlling condition of the while loop becomes false.

Ex:

Program 6.7 :

```
/*  FACTORIAL OF THE GIVEN NUMBER BY USING WHILE LOOP   */
#include<stdio.h>
main()
 {
    int x,i=1,rst=1;
    printf("Enter any integer number : ");
    scanf("%d",&x);
    while(i<=x)
    {
        rst=rst*i;
        i++;
    }
    printf("Factorial of %d is : %d",x,rst);
 }
```

OUTPUT:

```
Enter any integer number : 5
Factorial of 5 is : 120
```

Explanation : The above program to accepts an integer numer to calculate the factorial. The variable i is initialized with 1, till this I value becomes to given number the body of the while loop will execute. The factorial of the given number will saved in the variable rst and printed it.

6.9 DO-WHILE LOOP:

The do-while loop performs the test at the bottom rather than at the top. The do-while loop start with the keyword do, followed by the body of the loop.

The general format :

```
do
{
       statement 1;
       statement 2;
       statement 3;
            .
            .
       statement n;
```

```
}while(expression);
```

After executing the body of the loop it reaches the while, the expression specified is evaluated. If it is found to be true, automatically loop back to the top and re-enter the body of the loop. If at the time of testing, the condition evaluates as false, you break out the do-while loop.

Ex :

Program 6.8 :

```
/*  FACTORIAL OF THE GIVEN NUMBER BY USING DO_WHILE LOOP   */
#include<stdio.h>
main()

{
    int x,i=1,rst=1;
    printf("Enter any integer number : ");
    scanf("%d",&x);
    do
    {
          rst=rst*i;
          i++;
    }while(i<=x);
    printf("Factorial of %d is : %d",x,rst);
}
```

OUTPUT:

```
Enter any integer number : 5
Factorial of 5 is : 120
```

Explanation : The above program to accepts an integer numer to calculate the factorial. The variable i is initialized with 1, till this I value becomes to given number the body of the do-while loop will executes. The factorial of the given number will saved in the variable rst and printed it.

6.10 FOR LOOP :

This is used when the statements are to be executed more than once. This is the most widely used iteration construct. The for loop supported by C is much more powerful than its counterpart in other high level languages.

The general format:

```
for(initialization; expression; increment)

{
    statement 1;
    statement 2;
          .
          .
    statement n;
}
```

The for loop starts with for keyword. The keyword for is followed by a parenthesized with a header. This is followed by the body of the loop which typically is a set of statements enclosed between braces. The header of the for loop consists of 3 portions; first portion consists of a set of statements to be executed initially, second portion is an expression that will act as the controlling condition of the loop and final portion consists of incrementing or decrementing.

Ex:

Program 6.9 :

```
/*  FACTORIAL OF THE GIVEN NUMBER BY USING FOR LOOP   */
#include<stdio.h>
main()

{
    int x,i,rst=1;
    printf("Enter any integer number : ");
    scanf("%d",&x);
```

```
    for(i=1;i<=x;i++)
        rst=rst*i;
    printf("Factorial of %d is : %d",x,rst);
}
```

OUTPUT:

Enter any integer number : 5

Factorial of 5 is : 120

Explanation : The above program to accepts an integer numer to calculate the factorial. Within the for loop the variable i is initialized with 1, till this I value becomes to given number the body of the for loop will execute, with increments of 1. The factorial of the given number will saved in the variable rst and printed it.

Comma operator :

Comma operators are basically used in for loops to have more than one initialization and increment statements.

The general format :

```
for(exp1,exp2;exp3;exp4,exp5)
{
    statements;
}
```

6.11 BREAK & CONTINUE :

C provides two statements - break and continue using which the normal behavior of a loop can be altered. We already have used the break statement in switch statement. It can also be used inside a while loop, a for loop and do-while loop. It causes control to break out the innermost control structure. C does not provide any mechanism to break out of an outer enclosing loop. Because of its nature a break will always be conditional (attached to an if).

The general format :

```
while(1)
{
    /* do something */
    if(some condition)
        break;
    /* do something */
}
```

The continue statement whenever executed causes the rest of current iteration to be skipped and causes the next iteration to begin, subjecting of course to the truth of the controlling condition.

The general format :

```
while(exp)
{
    /* do something */
    if(some condition)
        continue;
    /* do something */
}
```

exit() : It is a standard library function used to terminate the program execution.

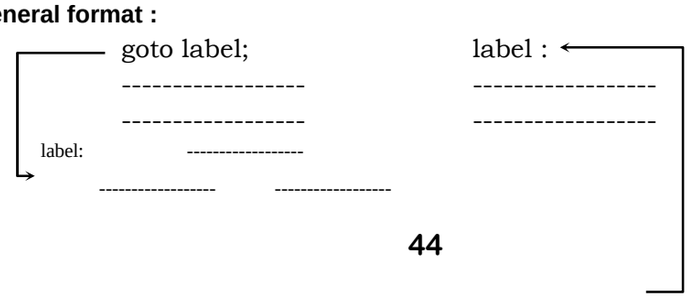
The general format :

```
    exit(argument);
```

goto : C supports the goto statement to branch unconditionally from one point to another in the program. Although it may not be essential to use the goto statement in a highly structured language like C, there may be occasions when the use of goto might be desirable.

The goto requires a label in order to identify the place where the branch is to be made. A label is any valid variable name, and must be followed by a colon. The label is placed immediately before the statement where the control is to be transferred.

The general format :



---------- goto label ;

The label: It can be anywhere in the program either before or after the goto label statement. If the label is before the statement goto label a loop will be formed and some statements will be executed repeatedly. Such a jump is known as backward jump. On the other hand if the label is placed after the goto label some statements will be skipped and the jump is known as a forward jump.

6.12 OBJECTIVE TYPE QUESTIONS

1. Identify the unconditional control structure: [ ]
   a) do-while b) switch-case c) goto d) if
2. Identify the loop construct: [ ]
   a) if-else b) switch-case c) goto d) while
3. The number of loop constructs in C is: [ ]
   a) 2 b) 3 c) 4 d) 5
4. Identify the wrong statement: [ ]
   a) if (a<b); b) if a<b; c) if (a<b) { ;} d) options b and c
5. which is syntactically correct: [ ]
   a) if (a:=10) {..} else if (a<10) {..}
   b) if (a==10){..} else if (a<10) { ..}
   c) if (a eq 10) { .. } else if (a<10) {..}
   d) if (a.eq. 10) {..} else if (a<10) {..}
6. Which is correct statement: [ ]
   a) printf("Maximum=%d",(x.y)?x:y);
   b) printf("%s",(marks >=60)? "first class" : "Not firstclass";
   c) printf("%s","pass"); d) all the above
7. The minimum number of times the while loop is executed is
   a) 0 b) 1 [ ]
   c) 2 d) cannot be predicted
8. The minimum number of times the for loop is executed is [ ]
   a) 0 b) 1
   c) 2 d) cannot be predicted
9. The minimum number of times the do-while loop is executed is [ ]
   a) 0 b)1 c) 2 d) cannot be predicted
10. Continue statement is used to [ ]
    a) continue the next iteration of a loop construct
    b) exit the block where is exists and continues further
    c) exit the outermost block even if it occurs inside the innermost
    d) continue the compilation even an error occurs in a program
11. The break statement is used in [ ]
    a) selective control structures only
    b) loop control structures only
    c) options a and b
    d) switch-case control structures only
12. Each case statement in switch() is separated by [ ]
    a) continue b) goto c) break d) None
13. The switch statement is used to [ ]
    a) switch from one variable to another
    b) choose from multiple possibilities which may arise due to different values of a single variable
    c) switch between functions in a program
    d) None of the above
14. What will be the value of x after execution of the program [ ]

```
#include<stdio.h>
main()
{
    int x=5,y=0;
    while(y<10)
    {
```

```
            printf("%d%d",y++,x++);
        }
    }
```

a) 5 b) 14 c) 15 d) 16

15. In the following code point out the error, if any in the *for* loop [ ]

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

 {
    int i=1;
    for( ; ; )
    {
            printf("%d",i++);
            if (i>5)
                break;
     }
 }
```

a) condition in the *for* loop is must
b) two semicolons should be dropped
c) for loop should be replace with other loop
d) No error

16. In the following code point out the error, if any in the *while* loop [ ]

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

 {
    int i=1;
    while( )
    {
            printf("%d",i++);
            if (i>5)
                break;
     }
 }
```

a) condition in the *while* loop is must
b) there should be ateast one semicolon
c) while loop should be replace with other loop
d) No error

18. In the following code, if c is a variable initialized to 2, the lowing loop executes..... number of times [ ]

```
while((c>0)&&(c<10))
{
        body of the loop
        c++;
}
```

a) 10 b) 8 c) 9 d)0

19. What is the functionality of the following program ? [ ]

```
main( )
{
      int  c=0;
      do{
            printf("%d\n",c++);
      }while(c <= 9);
}
```

a) adding 9 integers b) adding integers from 1 to 9
c) display any 9 integers d) display integers from 0 to 9

ANSWERS

1.	c	2.	d	3.	b	4.	b	5.	b
6.	d	7.	a	8.	a	9.	b	10.	a
11.	c	12.	c	13.	b	14.	b	15.	d
16.	a	17.	b	18.	D				

6.13 EXAMPLE PROGRAMS

Program 6.10 :

```
/*  PROGRAM TO ACCEPT A NUMBER AND FIND OUT IS IT EVEN OR ODD   */
#include<stdio.h>
main()

{
    int n;
    printf("Enter any number  :   ");
    scanf("%d",&n);
    if(n%2==0)
       printf(" \nThe given number is EVEN \n");
    else
       printf(" \nThe given number is ODD\n");
}
```

OUTPUT:

```
Enter any number  :   45
The given number is  ODD
```

Explanation : The above program to accepts an integer numer. It is checked, whether is it divided by 2 or not, based on that it will print EVEN or ODD.

Program 6.11 :

```
/*  PROGRAM TO ACCEPT THREE SIDES (POSITIVE INTEGERS) OF A TRIANGLE
DETERMINE WHETHER THEY FORM A VALID TRIANGLE  */
#include<stdio.h>
main()

{
    int a,b,c;
    printf("Enter first side of a triangle  :   ");
    scanf("%d",&a);
    printf("Enter second side of a triangle  :   ");
    scanf("%d",&b);
    printf("Enter third side of a triangle  :   ");
    scanf("%d",&c);
    if (a+b>c || a+c>b || b+c>a)
        printf("  It will form a valid triangle \n");
    else
        printf("  It will not form a valid triangle \n");
}
```

OUTPUT:

```
Enter first side of a triangle  :  4
Enter second side of a triangle  :  1
Enter third side of a triangle  :  6
It will form a valid triangle
```

Explanation : The above program accepts three integer numers. By using if condition it is checking three conditions. If any one condition is satisfied, it will form a valid triangle.

Program 6.12 :

```
/*   PROGRAM TO ACCEPT A CHARACTER AND FINDOUT IS IT UPPERCASE OR
LOWERCASE CHARACTER   */
#include<stdio.h>
main()

{
   int c;
   printf("Enter any character : ");
   c=getchar();
   if(c>97)
      printf("The given character is LOWERCASE character \n");
   else
      printf("The given character is UPPERCASE character \n");
```

```
}
```

OUTPUT:

Enter any character : r

The given character is LOWERCASE character

Explanation : The above program accepts an alphabet. This c value is checked by if codition. If it is greaterthan 97 it is Lowercase letter otherwise it is Uppercase. Lowercase alphabets ASCII value is greaterthn 97, uppercase alphabets ASCII value is lessthan 97 only.

Program 6.13 :

```
/*   PROGRAM TO FIND THE GIVEN CHARACTER IS VOWEL OR CONSONANT
*/
#include<stdio.h>
main()
{
    char c;
    printf("Enter any character : ");
    c=getchar();
if(c=='a'||c=='e'||c=='i'||c=='o'||c=='u'||c=='A'||c=='E'||c=='I'||c=='O'||c=='U')
        printf("  The given character is VOWEL \n");
    else
        printf("  The given character is CONSONENT \n");
}
```

OUTPUT:

Enter any character : u

the given character is VOWEL

Explanation : The above program accepts an alphabet. This c value is checked by compound conditions to identify the given alphabet is vowel or consonant.

Program 6.14 :

```
/*   PROGRAM TO GENERATE NATURAL NUMBERS TILL THE SPECIFIED
NUMBER */
#include<stdio.h>
main()
{
    int i,n;
    printf("Enter any number  : ");
    scanf("%d",&n);
    for(i=0;i<=n;i++)
       printf("%d  ",i);
}
```

OUTPUT:

Enter any number : 12

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

Explanation : The above program accepts an integer number as n. Within the for loop the integer variable i is initialized with 1 and incremented by 1 till this i value becomes the given number. Each time the i value will be printed.

Program 6.15 :

```
/*   PROGRAM TO GENERATE THE EVEN NUMBERS TILL THE SPECIFIED
NUMBER */
#include<stdio.h>
main()
{
    int i,n;
    printf("Enter any number  : ");
    scanf("%d",&n);
    for(i=2;i<=n;i+=2)
       printf("%d  ",i);
```

```
}
```

OUTPUT:

```
Enter any number  : 20
2  4  6  8  10  12  14  16  18  20
```

Explanation : The above program accepts an integer number as n. Within the for loop the integer variable i is initialized with 2 (starting een number) and incremented by 2 till this i value becomes the given number. Each time, the i value will be printed.

Program 6.16 :

```
/*    PROGRAM TO GENERATE EVEN NUMBERS DESCENDING ORDER SRARTING FROM THE SPECIFIED NUMBER  */
#include<stdio.h>
main()
{
    int n;
    printf("Enter any number  : ");
    scanf("%d",&n);
    if(n%2!=0)
      n--;
    while(n>0)
    {
      printf("%d  ",n);
      n-=2;
    }
}
```

OUTPUT:

```
Enter any number  : 20
20  18  16  14  12  10  8  6  4  2
```

Explanation : The above program accepts an integer number as n. Check the given is even or not, if not it will decremented by 1. Now n value is the starting even number, till this value becomes zero it will prints and decremented by 2.

Program 6.17 :

```
/*  PROGRAM TO ACCEPT TWO NUMBERS AND CALCULATE THE PRODUCT OF IT BY USING THE SUCCESSIVE ADDITION METHOD   */
#include<stdio.h>
main()
 {
      int i,a,b,rst=0;
      printf("Enter the first number i.e A   : ");
      scanf("%d",&a);
      printf("Enter the second number i.e B  : ");
      scanf("%d",&b);
      for(i=1;i<=b;i++)
            rst+=a;
      printf("%d   X   %d    =   %d\n",a,b,rst);
 }
```

OUTPUT:

```
Enter the first number i.e A   : 8
Enter the second number i.e B  : 9
8   X   9    =   72
```

Explanation : The above program accepts two integers numbers as a and b. The rst variable is initialized with zero. Variable a value will be added to rst variable b number of times.

Program 6.18 :

```
/*  PROGRAM TO ACCEPT TWO NUMBERS AND CALCULATE THE PRODUCT OF IT BY USING RUSSIAN TECHNIC METHOD   */
#include<stdio.h>
```

```
main()
{
      int x,y,i,a,b,rst=0;
      printf("Enter the first number i.e A   : ");
      scanf("%d",&a);
      printf("Enter the second number i.e B  : ");
      scanf("%d",&b);
      x=a;
      y=b;
      printf("\n\n\n\t   A              B \n\n");
      while (a>0)
      {
          if (a%2!=0)
          {
                printf("\t  %3d           %3d +\n",a,b);
                rst+=b;
          }
          else
                printf("\t  %3d           %3d \n",a,b);
          a=a/2;
          b=b*2;
      }
      printf("\n\n\t   %d      X      %d     =      %d\n",x,y,rst);
}
```

OUTPUT:

```
Enter the first number i.e A  : 23
Enter the second number i.e B : 10
            A             B
            23            10+
            11            20+
            5             40+
            2             80
            1            160+
            23 X 10 = 230
```

Explanation : The above program accepts two integers numbers as a and b. The rst variable is initialized with zero. If variable a value is odd value then b value will be added to rst. Variable a value will be divided by 2 and b value will be multiplied by 2. Till a value becomes zero it will repeat.

Program 6.19 :

```
/*     PROGRAM TO ACCEPT X AND Y VALUES AND CALCULATE THE X^Y  VALUE
*/
#include<stdio.h>
main()
{
    int i,x,y,rst=1;
    printf("Enter the base value i.e X  : ");
    scanf("%d",&x);
    printf("Enter the power value i.e Y  : ");
    scanf("%d",&y);
    for(i=1;i<=y;i++)
       rst*=x;
    printf("\n\n\t The   %d  power %d value is\t = %d\n",x,y,rst);
}
```

OUTPUT:

```
Enter the base value i.e X : 3
Enter the power value i.e Y : 3
the  3  power  3  value is  =  27
```

Explanation : The above program accepts two integers numbers as x and y. The rst variable is initialized with 1. Now rst will multiplied with x, y number of times.

Program 6.20 :

```
/* PROGRAM TO ACCEPT A NUMBER AND PRINT ITS REVERSE NUMBER   */
#include<stdio.h>
main()
{
    int t,n,r=0;
    printf("Enter any number  : ");
    scanf("%d",&n);
    t=n;
    while (n>0)
    {
        r=r*10+n%10;
        n=n/10;
    }
    printf("The reverse of the  %d  number   %d\n",t,r);
}
```

OUTPUT:

Enter any number : 4567

The reverse of the 4567 number 7654

Explanation : The above program accepts an integer number as n. The variable r is initialized with 1. This r will be multiplied with 10 and add the right most digit of the given number (n%10). Now n will n/10, i.e. right ost digit will be deleted. This process will continue till n becomes zero. Now r will be the reverse of the given number.

Program 6.21 :

```
/*   PROGRAM TO ACCEPT A NUMBER AND FIND OUT IS IT ARMSTRONG
NUMBER OR NOT  */
#include<stdio.h>
main()
{
    int t,n,r=0;
    printf("Enter any number  : ");
    scanf("%d",&n);
    t=n;
    while (n>0)
    {
        r=r+(n%10*n%10*n%10);
        n=n/10;
    }
    if (t==r)
       printf(" The number   %d   is ARMSTRONG number\n",t);
    else
       printf(" The number   %d   is NOT ARMSTRONG number\n",t);
}
```

OUTPUT:

Enter any number : 432

The number 432 is NOT ARMSTRONG number

Explanation : The above program accepts an integer number as n. The variable r is initialized with zero. This r will be added to the right most digit of the given number (n%10). Now n will n/10, i.e. right ost digit will be deleted. This process will continue till n becomes zero. This r value will copared with the given number, if both are same it is Armstrong number otherwise not Armstrong number.

Program 6.22 :

```
/*  PROGRAM TO ACCEPT A NUMBER AND FIND OUT WHETHER IS IT
    POLINDROME OR NOT   */
#include<stdio.h>
main()
{
```

```
    int t,n,r=0;
    printf("Enter any number  : ");
    scanf("%d",&n);
    t=n;
    while (n>0)
    {
        r=r*10+n%10;
        n=n/10;
    }
    if(r==t)
        printf("The number %d is PALINDROME\n",t);
    else
        printf("The number %d is NOT PALINDROME\n",t);
}
```

OUTPUT:

Enter any number : 6556
The number 6556 is PALINDROME

Explanation : The above program accepts an integer number as n. The variable r is initialized with 1. This r will be multiplied with 10 and add the right most digit of the given number (n%10). Now n will n/10, i.e. right ost digit will be deleted. This process will continue till n becomes zero. Now r will be the reverse of the given number. This reverse number r will be compared with the given number. If both are same then the given number is Palindrome otherwise not Palindrome.

Program 6.23 :

```
/*   PROGRAM TO ACCEPT TWO NUMBERS AND CALCULATE THE GCD OF IT
*/
#include<stdio.h>
main()
{
   int x,y,i,a,b;
   printf("Enter the first number i.e A   : ");
   scanf("%d",&a);
   printf("Enter the second number i.e B  : ");
   scanf("%d",&b);
   x=a;
   y=b;
   while(a!=b)
   {
      if(a>b)
         a=a-b;
      else
         b=b-a;
   }
   printf("GCD of %d & %d  is : %d\n",x,y,a);
}
```

OUTPUT:

Enter the first number i.e A : 45
Enter the second number i.e B : 60
GCD of 45 & 60 is : 15

Explanation : The above program accepts two integer numbers as a and b. If both values are not equal, the lowest value will be subtracted form highest value. This process will repeat till both values becomes equal. Finally it will prints either a or b value i.e. GCD of given two numbers.

Program 6.24 :

```
/*   PROGRAM TO ACCEPT TWO NUMBERS AND CALCULATE THE LCM OF IT
*/
#include<stdio.h>
main()
{
   int x,y,i,a,b,l;
   printf("Enter the first number i.e A   : ");
   scanf("%d",&a);
   printf("Enter the second number i.e B  : ");
   scanf("%d",&b);
   x=a;
   y=b;
   while(a!=b)
   {
      if(a>b)
         a=a-b;
      else
         b=b-a;
   }
   l=(x*y)/a;
   printf("LCM of %d & %d  is : %d\n",x,y,l);
}
```

OUTPUT:

Enter the first number i.e A : 25

Enter the second number i.e B : 20

LCM of 25 & 20 is : 100

Explanation : The above program accepts two integer numbers as a and b. If both values are not equal, the lowest value will be subtracted form highest value. This process will repeat till both values become equal. Now the product of given two numbers will be divided by either a or b and stored in the variable l. The value of the l is LCM of given two numbers.

Program 6.25 :

```
/* PROGRAM TO ACCEPT A NUMBER AND FIND OUT WHETHER IS IT PRIME OR
NOT  */
#include<stdio.h>
main()
{
   int i,n,c=0;
   printf("Enter any number  : ");
   scanf("%d",&n);
   i=1;
   while(i<=n)
   {
      if (n%i==0)
         c++;
      i++;
   }
   if(c>2)
        printf(" The number  %d  is NOT PRIME NUMBER\n",n);
   else
        printf(" The number  %d  is PRIME NUMBER\n",n);
}
```

OUTPUT:

Enter any number : 11

The number 11 is PRIME NUMBER

Explanation : The above program accepts two integer numbers as a and b. If both values are not equal, the lowest value will be subtracted form highest value. This process will repeat till both values become equal. Finally it will prints either a or b value i.e. GCD of given two numbers.

Program 6.26 :

```
/* PROGRAM TO PRINT MULTIPLICATION TABLE OF THE SPECIFIED NUMBER */
#include<stdio.h>
main()
{
   int i,n,c=0;
   printf("Enter any number  : ");
   scanf("%d",&n);
    i=1;
   while(i<=10)
   {
      printf("  %d  X  %d  =  %d\n",n,i,n*i);
      i++;
   }
}
```

OUTPUT:

```
Enter any number : 12
    12 X  1  = 12
    12 X  2  = 24
    12 X  3  = 36
    12 X  4  = 48
    12 X  5  = 60
    12 X  6  = 72
    12 X  7  = 84
    12 X  8  = 96
    12 X  9  = 108
    12 X  10 = 120
```

Explanation : The above program accepts an integer number as n. It prints the multiplication table of the given number by using while loop.

Program 6.27 :

```
/*  PROGRAM TO PRINT ALL LOWER CASE LETTERS FROM THE SPECIFIED CHARACTER TO SPECIFIED CHARACTER    */
#include<stdio.h>
main()
{
   int n;
   char f,l;
   clrscr();
   printf(" Enter a character from where you start : ");
   f=getchar();
   getchar();
   printf(" Enter a character where to stop : ");
   l=getchar();
   puts(" The letters are :");
   for(n=f;n<=l;n++)
       printf(" %c  ",n);
}
```

OUTPUT:

```
Enter a character from where you start : d
Enter a character where to stop : l
The letters are :
      d  e  f  g  h  i  j  k  l
```

Explanation : The above program accepts two characters i.e. starting and ending characters. In the for loop n is initialized with the starting character, prints value of n and it is incrementing by 1 till the ending character.

Program 6.28 :

```
/*  PROGRAM TO GENERATE THE FIBONACY NUMBERS TILL THE SPECIFIED
NUMBER  */
#include<stdio.h>
main()

{
   int a=0,b=1,n,c;
   printf("Enter any number  : ");
   scanf("%d",&n);
   printf("The fibonacy numbers are  . . . .:");
   printf("\n%d   %d   ",a,b);
   c=a+b;
   while (c<=n)
   {
       printf("%d  ",c);
       a=b;
       b=c;
       c=a+b;
   }
   printf("\n");
}
```

OUTPUT:

```
Enter any number  : 100
The fibonacy numbers are  .  .  .  .  :
1  1  2  3  5  8  13  21  34  55  89
```

Explanation : The above program accepts an integer number n. The variables a and b is initialized with the 0 and 1 and prints. Next number will be identified as addition of a and b will stored in c. Till c value becomes greater than or equal to n, it will prints c and next c can be calculated.

Program 6.29 :

```
/*  PROGRAM TO ACCEPT A NUMBER N, FINDS AND DISPLAYS THE SUM OF
INTEGERS    1 TO 2, 1 TO 3, 1 TO 4, . . . , 1 TO N   */
#include<stdio.h>
main()

{
   int i,m,t,n,sum;
   printf("Enter any number  : ");
   scanf("%d",&m);
   for(i=2;i<=m;i++)
   {
      sum=0;
      t=i;
      do
      {
         sum+=i;
         i--;
      }while(i>=1);
      printf("%d  ",sum);
      i=t;
   }
}
```

OUTPUT:

```
Enter any number : 5
3   6   10   15
```

Explanation : The above program accepts an integer number m. By using nested loops, here within the for loop we are using do-while loop to perform the above operation.

Program 6.30 :

```
/*  PROGRAM TO PRINT 1!,2!,3!,4!,. . . ,N!  */
#include<stdio.h>
main()
{
    int i,j,n,fact;
    printf("Enter any number  : ");
    scanf("%d",&n);
    for(i=1;i<=n;i++)
    {
        fact=1;
        for(j=1;j<=i;j++)
            fact*=j;
        printf("%d,   ",fact);
    }
}
```

OUTPUT:

```
Enter any number : 5
1,   2,   6,   24,   120,
```

Explanation : The above program accepts an integer number m. By using nested loops, here within the for loop we are using another for loop to perform the above operation.

Program 6.31 :

```
/*  PROGRAM TO GENERATE PRIME NUMBERS TILL THE SPECIFIED NUMBER
*/
#include<stdio.h>
main()
{
    int j,i,n,c;
    printf("Enter any number  : ");
    scanf("%d",&n);
    for(j=1;j<=n;j++)
    {
        c=0;
        i=1;
        while(i<=j)
        {
          if (j%i==0)
             c++;
          i++;
        }
        if(c<=2)
           printf("%d  ",j);
    }
}
```

OUTPUT:

```
Enter any number : 10
1   2   3   5   7
```

Explanation : The above program accepts an integer number n. By using nested loops, here within the for loop we are using while loop to perform the above operation.

Program 6.32 :

```
/* PRINT THE FOLLOWING FORMAT
   1
   1  2
   1  2  3
   1  2  3  4
   1  2  3  4  5
   . . . . . .
*/
```

```
#include<stdio.h>
main()
{
   int n,i,j;
   clrscr();
   printf("Enter an Integer number : ");
   scanf("%d",&n);
   for(i=1;i<=n;i++)
   {
       for(j=1;j<=i;j++)
           printf("%d  ",j);
       printf("\n");
   }
}
```

Explanation : The above program accepts an integer number n. By using nested loops, here within the for loop we are using another for loop to print the above format.

Program 6.33 :

```
/* PRINT THE FOLLOWING FORMAT
  1
  2  1
  3  2  1
  4  3  2  1
  5  4  3  2  1
  . . . . . .
*/
#include<stdio.h>
main()
{
   int n,i,j;
   clrscr();
   printf("Enter an Integer number : ");
   scanf("%d",&n);
   for(i=1;i<=n;i++)
   {
       for(j=i;j>=1;j--)
           printf("%2d  ",j);
       printf("\n");
   }
}
```

Explanation : The above program accepts an integer number n. By using nested loops, here within the for loop we are using another for loop to print the above format.

Program 6.34 :

```
/* PRINT THE FOLLOWING FORMAT
                  1
              2     2
          3     3     3
      4     4     4     4
  5     5     5     5     5
  .     .     .     .     .
*/
#include<stdio.h>
main()
{
   int n,i,k,j;
   clrscr();
   printf("Enter an Integer number : ");
   scanf("%d",&n);
```

```
   for(i=1;i<=n;i++)
   {
      for(k=1;k<=40-4*i/2;k++)
         printf(" ");
      for(j=i;j>=1;j--)
         printf("%4d",i);
      printf("\n");
   }
}
```

Explanation : The above program accepts an integer number n. By using nested loops, here within the for loops we are using another for loop to print the above format.

Program 6.35 :

```
/* PRINT THE FOLLOWING FORMAT
                              1
                         2    2
                    3    3    3
               4    4    4    4
          5    5    5    5    5
               .   .   .   .   .
*/
#include<stdio.h>
main()

{
   int n,i,k,j;
   clrscr();
   printf("Enter an Integer number : ");
   scanf("%d",&n);
   for(i=1;i<=n;i++)

   {
      for(k=1;k<=40-3*i;k++)
         printf(" ");
      for(j=i;j>=1;j--)
         printf("%3d",i);
      printf("\n");
   }
}
```

Explanation : The above program accepts an integer number n. By using nested loops, here within the for loops we are using another for loop to print the above format.

Program 6.36 :

```
/*  PRINTS THE GIVEN THREE NUMBERS IN ASCENDING ORDER   */
#include<stdio.h>
#include<conio.h>
/*    Main program begins    */
main()
{
       int a,b,c;
       clrscr();
       /*   Accepting inputs from user    */
       printf("\n Enter  'a' value  :  ");
       scanf ("%d",&a);
       printf("\n Enter  'b' value  :  ");
       scanf("%d",&b);
       printf("\n Enter  'c' value  :  ");
       scanf("%d",&c);
```

```
    /*   Evaluation and print in ascending order  */
    if(a<=b&&a<=c)
    {
        if(b<c)
            printf(" The given numbers in ascending order is: %d, %d, %d \n",a,b,c);
        else
            printf("The given numbers in ascending order is:%d,%d,%d \n",a,c,b);
    }
    if(b<=a&&b<=c)
    {
    if(a<c)
            printf("The given numbers in ascending order is :  %d,  %d,  %d \n",b,a,c);
    else
            printf("The given numbers in ascending order is: %d,  %d,  %d \n",b,c,a);
    }
    if(c<=a&&c<=b)
    {
            if(a<b)
            printf("The given numbers in ascending order is: %d, %d, %d \n",c,a,b);
   else
            printf("The given numbers in ascending order is: %d,  %d,  %d \n",c,b,a);
    }
}
```

OUTPUT:

Enter 'a' value : 12
Enter 'b' value : 10
Enter 'c' value : 2
The given numbers in ascending order is : 2, 10, 12

Explanation : The above program accepts three integer numbers. By using compound if – else statements we can print the given three numbers in ascending order.

Program 6.37 :

```
/* C PROGRAMS TO PRINT THE FOLLOWING OUTPUTS USING FOR LOOP.
    1                                       1
    2   2                                 2   2
    3   3   3                           3   3   3
    4   4   4   4                     4   4   4   4
    5   5   5   5   5               5   5   5   5   5
*/
```

i) Program :

```
#include<stdio.h>
main()
{
    int n,i,j;
    clrscr();
    printf("Enter an Integer number : ");
    scanf("%d",&n);
    for(i=1;i<=n;i++)
    {
        for(j=1;j<=i;j++)
        printf("%d  ",i);
        printf("\n");
    }
}
```

ii) Program :

```
#include<stdio.h>
main()
{
        int n,i,k,j;
        clrscr();
        printf("Enter an Integer number : ");
        scanf("%d",&n);
        for(i=1;i<=n;i++)
        {
                for(k=1;k<=40-4*i/2;k++)
                        printf(" ");
                for(j=i;j>=1;j--)
                        printf("%4d",i);
                printf("\n");
        }
}
```

Program 6.38 :

```
/*  PROGRAM TO GENERATE SPECIFIED NUMBER OF FIBONACY NUMBERS  */
#include<stdio.h>
main()
{
        int a=0,b=1,n,c,i=2;
        printf("Enter how many Fibonaci numbers do you want  :  ");
        scanf("%d",&n);
        printf("The fibonacy numbers are  . . . .:");
        printf("\n%d   %d   ",a,b);
        c=a+b;
        do
        {
                printf("%d  ",c);
                i++;
                a=b;
                b=c;
                c=a+b;
        }while(i<=n);
        printf("\n");
        getchar();
}
```

OUTPUT:

```
Enter any number  : 10
The fibonacy numbers are  .  .  .  .  :
1  1  2  3  5  8  13  21  34  55
```

Program 6.39 :

/* A cloth show room has announced the following seasonal discounts on purchase of items.

PURCHASE	Discount (percentage)	
AMOUNT	Mill Cloth	Handloom Items
1-100	---	5.0
101-200	5.0	7.5
201-300	7.5	10.0
Above 300	10.0	15.0

Write a C program using switch and if statements to complete the net amount to be paid by a customer. */

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

```
#include<math.h>
/*    Main program begins    */
main()
{
        int amt, cho, item=0;
        float finalamt, dis=0.0;
        clrscr();
        /*   Accepting inputs from user    */
        printf("\n Enter amount  :  ");
        scanf ("%d",&amt);
        printf(" Enter item number 1-Mill cloth and 2-Handloom cloth: ");
        scanf("%d",&item);
        /*  Discount Evaluation   */
        switch(item)
        {
              case  1:
                    if (amt>=1 && amt<=100)
                          dis=0.0;
                    else if(amt>=101 && amt<=200)
                          dis=5.0;
                    else if(amt>=201 && amt<=300)
                                dis=7.5;
                          else
                                dis=10.0;
                    break;
           case  2:
                    if (amt>=1 && amt<=100)
                          dis=5.0;
                    else if(amt>=101 && amt<=200)
                              dis=7.5;
                    else if(amt>=201 && amt<=300)
                                dis=10.0;
                           else
                                dis=15.0;
                    break;
           default:
                    printf(" Invalid Cloth Selection  \n");
                    exit(0);
        }
        finalamt=amt-(dis/100.0)*amt;
        /*  Print the Result   */
        printf(" Discount is  :  %f  ",dis);
        printf(" Total amount is  :  %f  ",finalamt);
        getch();
}
```

OUTPUT:

```
Enter amount  :  200
Enter item number  1 -  Mill cloth  and  2  -  Handloom cloth  :  1
Discount is  :  5.000000
Total amount is  :  190.000000
```

Program 6.40 :

```
/*   PROGRAM TO PERFORM COMPLEX NUMBER  OPERATIONS   */
#include<stdio.h>
#include<conio.h>
/*    Main program begins    */
float ans1,ans2;
void main()
```

```
{
    /*   functions declaration    */
    void sum(int a, int b, int c, int d);
    void sub(int a, int b, int c, int d);
    void mul(int a, int b, int c, int d);
    void div(int a, int b, int c, int d) ;
    int a,b,c,d,cho;
    char sign;
    do
    {
        clrscr();
        /*  Displaying the Menu   */
        printf("\n\t 1  -   Addition of complex Numbers");
        printf("\n\t 2  -   Subtraction of complex Numbers");
        printf("\n\t 3  -   Multiplication of Complex Numbers");
        printf("\n\t 4  -   Division of  Complex Numbers");
        printf("\n\t 5  -   Exit");
        printf("\n\t\t\t  Enter your Choice  :  ");
        scanf("%d",&cho);
        if(cho>=5||cho<1)
        {
            printf("\n\t   No operation performed   \n");
            getch();
            exit(0);
        }
        /*   Accepting values from the user    */
        printf("\n Enter first complex number values (a+ib)  : \n");
        printf("\n Enter 'a' value  :  ");
        scanf(" %d",&a);
        printf("\n Enter 'b' value  :  ");
        scanf("%d",&b) ;
        printf("\n Enter second complex number values (c+id):\n");
        printf("\n Enter 'c' value  :  ");
        scanf("%d",&c) ;
        printf("\n Enter 'd' value  :  ");
        scanf("%d",&d) ;
        /*   Perform the specified operation   */
        switch(cho)
        {
            case 1:
                sum(a,b,c,d);
                sign='+';
                break;
            case 2:
                sub(a,b,c,d);
                sign='-';
                break;
            case 3:
                mul(a,b,c,d);
                sign='*';
                break;
            case 4:
                div(a,b,c,d);
                sign='/';
                break;
            default:
                printf("No complex operation \n");
                break;
        }
```

```
            /* Print the result */
            printf("\n\t(%d+i(%d)) %c (%d+i(%d))= (%f+i(%f))",a,b,sign,c,d,ans1,ans2);
            getch();
        }while(cho!=5);
}
/*   Function to addition of two complex numbers  */
void sum(int a, int b,int c,int d)
{
        ans1=a+c;
        ans2=b+d;
}
/*   Function to subtraction of two complex numbers  */
void sub(int a, int b,int c,int d)
{
        ans1=a-c;
        ans2=b-d;
}
/*   Function to multiply the two complex numbers  */
void mul(int a, int b, int c, int d)
{
        ans1=(a*c-(b*d));
        ans2=(a*d)+(b*c);
}
/*   Function to division of two complex numbers  */
void div(int a, int b,int c,int d)
{
        ans1=((a*c)+(b*d))/((a*a)+(b*b));
        ans2=((a*d)+(b*c))/((a*a)+(b*b));;
}
```

OUTPUT:

```
1  -    Addition of complex Numbers");
2  -    Subtraction of complex Numbers");
3  -    Multiplication of Complex Numbers");
4  -    Division of  Complex Numbers");
5  -    Exit");
Enter your Choice  :  1
Enter first complex number values (a+ib)  :
Enter 'a' value  :  1
Enter 'b' value  :  2
Enter second complex number values (c+id)  :
Enter 'c' value  :  4
Enter 'd' value  :  5
(1+i (2)) + (4+i(5)) = (5.000000 + i(7.000000));
```

Program 6.41 :

```
/*  Write C program using FOR statement to find the following from a given set of 20 integers
        I.   Total number of even integers
        II.  Total number of odd integers
        III. Sum of all even integers
        IV.  Sum of all odd integers    */

#include<stdio.h>
#include<conio.h>
/*    Main program begins    */
main()
{
        int n, e=0, o=0, es=0, os=0, i, m;
        clrscr();
        printf(" Enter how many numbers do you want  :  “);
        scanf(“%d”,&n);
```

```
        for(i=1;i<=n;i++)
        {
                printf(“  Enter the %dth number  :  “);
                scanf("%d",&m);
                if(m%2==0)
                {
                        e++;
                        es=es+m;
                }
                else
                {
                        o++;
                        os=os+m;
                }
        }
        /*      Print the specified values    */
        printf("\n\n The no of even numbers are  :  %d",e);
        printf("\n\n The no of odd numbers are  :  %d",o);
        printf("\n\n The sum of even numbers are  :  %d",es);
        printf("\n\n The sum of odd numbers are  :  %d",os);
        getch();
}
```

OUTPUT:

```
Enter how many numbers do you want  :  10
Enter the 1th number  :  5
Enter the 2th number  :  6
Enter the 3th number  :  7
Enter the 4th number  :  8
Enter the 5th number  :  9
Enter the 6th number  :  10
Enter the 7th number  :  11
Enter the 8th number  :  12
Enter the 9th number  :  13
Enter the 10th number  :  14
The no of even numbers are  :  5
The no of odd numbers are  :  5
The sum of even numbers are  :  50
The sum of odd numbers are  :  45
```

Program 6.42 :

```
/*      Program that will read the value of the following function :
                        1       for    x >0
                y = {   0       for    x = 0
                        -1      for    x < 0
        using   (i)     if statements
                (ii)    else if statements
                (iii)   conditional operator.    */
```

i) if statements

```
#include<stdio.h>
#include<conio.h>
/*    Main program begins    */
main()
{
        int y;
        float x;
        clrscr();
        /*   Accepting inputs from user    */
        printf("\n Enter  ‘x’ value   :  ");
        scanf ("%f",&x);
        if (x>0.0)
```

```
        y=1;
    if (x<0.0)
        y=-1;
    if (x==0)
        y=0;
    /* Print the Result */
    printf ("\n\t The value of y is : %d\n",y);
    getch();
}
```

OUTPUT:

```
Enter 'x' value : 0.0
The value of y : 0
```

ii) else-if statements

```
#include<stdio.h>
#include<conio.h>
/*  Main program begins  */
main()
{
    int y;
    float x;
    clrscr();
    /*  Accepting inputs from user  */
    printf("\n Enter 'x' value : ");
    scanf ("%f",&x);
    if (x>0.0)
        y=1;
    else if (x<0.0)
            y=-1;
        else
            y=0;
    /* Print the Result */
    printf ("\n\t The value of y is : %d\n",y);
    getch();
}
```

OUTPUT:

```
Enter 'x' value : -45.5
The value of y : -1
```

iii) Conditional operator

```
#include<stdio.h>
#include<conio.h>
/*  Main program begins  */
main()
{
    int y;
    float x;
    clrscr();
    /*  Accepting inputs from user  */
    printf("\n Enter 'x' value : ");
    scanf ("%f",&x);
    y=x>0.0?1(x<0.0?-1:0);
    /* Print the Result */
    printf ("\n\t The value of y is : %d\n",y);
    getch();
}
```

OUTPUT:

```
Enter 'x' value : 45.5
The value of y : 1
```

Program 6.43 :

```
/* PROGRAM TO FIND THE SUM OF 1 + 2 + 3 + . . . + N USING WHILE LOOP */
#include<stdio.h>
#include<conio.h>
/*  Main program begins  */
main()
{
      int sum=0,n,I=1;
      clrscr();
      /*  Accepting inputs from user  */
      printf("\nEnter  number of terms do you want  :  ");
      scanf ("%d",&n);
      /*  Calculating sum of first n numbers  */
      while(i<=n)
      {
            sum=sum+i;
            i++;
      }
      /*    Printing of answer    */
      printf("\n\n sum of the first %d numbers  is  :  %d\n",n,sum);
      getch();
}
```

OUTPUT:

Enter number of terms do you want : 10
sum of first 10 numbers is : 55

Program 6.44 :

/* PROGRAM TO EVALUATE THE SERIES $1+X^2/2!+X^4/4!+. . . . .$ UPTO 10 TERMS, ASSUME SUITABLE VALUE OF X */

```
#include<stdio.h>
#include<conio.h>
#include<math.h>
/*  Main program begins  */
main()
{
      int x, f=1, i, j, p=0;
      float y=0.0;
      clrscr();
      /*  Accepting inputs from user  */
      printf("\n Enter  'x' value  :  ");
      scanf ("%d",&x);
      /* Evaluation of series */
      for(i=0;i<10;i++)
      {
            for(j=1;j<=p;j++)
                  fact=fact*j;
            y=y+pow(x,p)/fact;
            p=p+2;
      }
      /*    Print the Result  */
      printf("\n\n The value of Y  is  :  %f\n",y);
      getch();
}
```

OUTPUT:

Enter 'x' value : 4
The value of Y is : 12.201101

Program 6.45 :

```
/*  PROGRAM TO EVALUATE THE SERIES (SIN X)  X-X3/3!+X5/5!+. . . . . UPTO 7
TERMS, ASSUME SUITABLE VALUE OF X.  */
#include<stdio.h>
#include<conio.h>
#include<math.h>
/*   Main program begins   */
main()
{
      int d,n;
      float x,y,term,sinx,acc=0.0000001;
      clrscr();
      /*  Accepting inputs from user   */
      printf("\nEnter 'x' value (in degrees)  :  ");
      scanf ("%f",&x);
      /*   conversion of degrees into radians   */
      y=x*3.1415/180;
      term=y;
      sinx=term;
      n=1;
      /*   Evaluation the expression  */
      do
      {
            d=2*n*(2*n+l);
            term=-term*y*y/d;
            sinx=sinx+term;
            n=n+l;
      }while(acc<=abs(sinx));
      /*   Print the result    */
      printf('\n\n  Sin(%f)  is   :   %f", x, sinx);
      getch();
}
```

OUTPUT:

```
Enter 'x' value (in degrees)  :  45
Sin(45.000000) is  :   0.707090
```

7 – FUNCTIONS

7.1 INTRODUCTION :

Functions act as the fundamental building blocks of large program. Each function normally performs a specific task. Every C program must contain at least one function named as main where the program always begins execution. The main function may call other functions with in it.

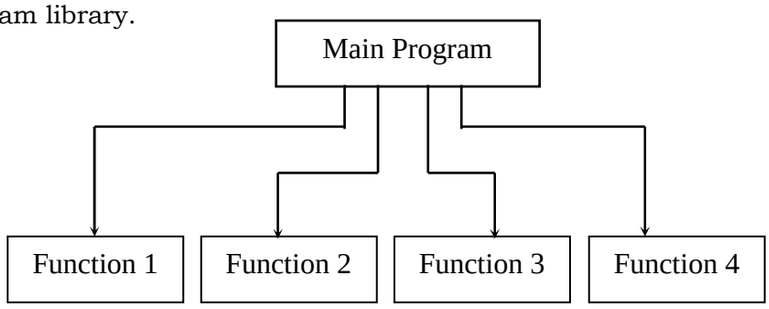
Importance of functions :

If a programmer can identify whatever actrions are performed repeatedly in various part of the program, write them as a function and call the function from the various parts of the program. This approach avoids code duplication. Instead of duplicating the same code in different parts of the program, you just write the function once and call it from different parts of the program, thus reducing the executable code size.

Breaking down your program into subroutines lends your program a structure. You can divide your task into multiple sub-tasks, develop a function for each sub-task, and integrate them into a single program. Once a function is written and tested, a programmer can expect it to fit with the rest of the program modules. This approach encourages code reusability.

7.2 USER DEFINED FUNCTIONS :

C functions can be classified into two categories namely library functions and user defined functions. Main is an example of user defined functions, printf, scanf, sqrt, etc,. are examples of library functions. The main difference between these two categories is that library functions are not written by the user, where as a user defined functions has to be developed by the user, at the time of writing a program. How ever, a user defined function can later become a part of the "C" program library.



The C function have three parts, they are function declaration, function call and function body.

Function declaration : A function can have a type. C allows bi-directional transmission of information between the caller of a function and the called function. The caller can communicate with the function by passing parameters, whereas the called function can communicate with the caller by returning a value. The caller will receive several parameters as many as you please. But the called function can return at the most only one value of only one type. So the type of a function specifies the type of the value that the function will return. Te general form :

```
type function_name(parameter list);
```

The function_name is must for every function. It may be combination of alphabets, digits and underscore, first character must be an alphabet, no special symbols other than underscore.

Function call : Calling a function is straight forward, call it by name and by supplying a parenthesized set of parameters as in.

```
value = function_name(par1, par2, . . .);
```

Function Body : The general form of any C function body is :

```
type function_name(parameter list)
parameters declaration
{
        local variable declarations;
        ---------------;
        ---------------;
        return(expression);
}
```

Parameter passing : Parameters are nothing but input given to a function. By passing parameters the caller can ask the function to process a set of values.

Parameter passing allows you to write generalized and re-usable functions. Whatever parameters the caller passes are called the *actual parameters* and whatever parameters the function has received are called the *formal parameters*. The actual parameter values are copied to the formal parameters.

Return value and their types : A function may or may not send back a value to the calling function. If it does, it is done through the **return** statement. The return statement can be any one of the following forms.

```
return;
  (or)
return(expression);
```

The first **return** statement does not return any value, it acts much as the closing brace of the function, when return is encountered the control is immediately passed back to the calling function. The second **return** statement return the value of the expression.

Calling a function : A function can be called by specifying the function name in a statement. When compiler encounters a function call, the control is transferred to that function. Once the function execution completes the control back to the calling function.

Local variables : A variable declared inside a function is called local variable. The scope of local variables is limited to the functions in which they are declared, or in other words these variables are inaccessible outside of this function. Like wise the scope of block variables is limited to that block in which they are declared.

Global variables : The variables you declare in the global variable section are called Global variables or External variables. While a local variable can be used only inside the function in which it is declared, a global variable can be used anywhere in the program. Global variables have a scope that spans the entire source program, because they can be used in any function.

Global vs Local variables :

1. Local variables can be used only inside the function or the block in which they are declared. On the other hand global variables can be used throughout the program.
2. All global variables, in the absence of explicit initialization, are automatically initialized to zero, i.e. int initialized with the value 0, float gets initialized with 0.0, char holds the ASCII null byte and pointer initialized with NULL. Local variables do not get initialized to any specific value. Thus a local variable starts up with an unknown value (garbage value), which may be different each time.
3. Global variables get initialized only once, typically just before the program starts executing. But, local variables get initialized each time the function or block gets called.
4. The initial value that you supplied for a global variable must be a constant, where as a local variable can contain variables in its initializer.
5. A local variable loses its value the movement the function/block containing it is exited. So you cannot expect a local variable to retain. Global variables retain there values through out the program's execution.

Scope of variables : The scope of local variables is limited to the functions in which they are declared, or in other words these variables are inaccessible outside of this function. Like wise the scope of block variables is limited to the block in which they are declared. Global variables have a scope that spans the entire source program, because they can be used in any function.

7.3 TYPES OF FUNCTIONS :

Function are cateforized by depending on whether arguments are present or not or whether a value is returned or not.

1. Function with no argument and no return values
2. Function with arguments and no return values
3. Function with arguments and return values

Function with no arguments and no return value : In this case, the functions do not contain any arguments and do not receive any data from the calling function. The following example program demonstrates this feature.

Program 7.1 :

```
/*  FUNCTION WITH NO ARGUMENTS AND NO RETURN VALUES  */
#include<stdio.h>
main()
{
```

```
        lineprint();
        power();
        lineprint();
}
lineprint()
{
        int i;
        for(i=0;i<=50;i++)
             printf("-");
        printf("\n");
}
power()
{
        int x,y,i,r;
        printf("      Enter the base value : ");
        scanf("%d",&x);
        printf("      Enter the power value : ");
        scanf("%d",&y);
        r=1;
        for(i=0;i<y;i++)
            r=r*x;
        printf("      %d power %d is : %d \n",x,y,r);
}
```

OUTPUT :

```
--------------------------------------------------
      Enter the base value :  2
      Enter the power value :  3
      2 power 3 is : 8
--------------------------------------------------
```

Explanation : In the above program there are two functions names printline and power. The printline function will print a line by using '-' character. The power function will accepts base and powere values from user as x and y will calculate x^y value. Both functions are not returning any value to main.

Function with Arguments and no return value : In this case, the function will accept the argument values from the user and send to and does not contain any return values. The following example program demonstrates this feature

Program 7.2 :

```
/* FUNCTION WITH ARGUMENTS AND NO RETURN VALUES    */
#include<stdio.h>
main()
{
        char c;
        int x,y;
        printf("Enter a character : ");
        c=getchar();
        lineprint(c);
        printf("      Enter the base value : ");
        scanf("%d",&x);
        printf("      Enter the power value : ");
        scanf("%d",&y);
        power(x,y);
        lineprint(c);
}
lineprint(ch)
char ch;
{
        int i;
        for(i=0;i<=50;i++)
                printf("%c",ch);
        printf("\n");
}
power(a,b)
int a,b;
```

```
{
        int i,r;
        r=1;
        for(i=0;i<b;i++)
                r=r*a;
        printf("      %d power %d is : %d \n",a,b,r);
}
```

OUTPUT :

```
Enter a character : *
************************************************
      Enter the base value :  2
      Enter the power value :  3
      2 power 3 is : 8
************************************************
```

Explanation : In the above program there are two functions names printline and power. The printline method will accept a character as parameter and print a line by using that character. The power function will accepts base and power values as parameters and will calculate x^y value. Both functions are not returning any value to main.

Function with Arguments and return values : In this case, the functions will accept argument vaues from the user and contains the return values. The following example program demonstrates this feature

Program 7.3 :

```
/*   FUNCTION WITH ARGUMENTS AND RETURN VALUES    */
#include<stdio.h>
main()
{
        char c;
        int x,y;
        printf("      Enter a character : ");
        c=getchar();
        printline(c);
        printf("      Enter the base value : ");
        scanf("%d",&x);
        printf("      Enter the power value : ");
        scanf("%d",&y);
        printf("      %d power %d is : %d \n",x,y,power(x,y));
        printline(c);
}
printline(ch)
char ch;
{
        int i;
        for(i=0;i<=50;i++)
                printf("%c",ch);
        printf("\n");
}
power(a,b)
int a,b;
{
        int i,r;
        r=1;
        for(i=0;i<b;i++)
                r=r*a;
        return(r);
}
```

OUTPUT :

```
Enter a character : *
************************************************
      Enter the base value :  2
      Enter the power value :  3
      2 power 3 is : 8
```

****

Explanation : In the above program there are two functions names printline and power. The printline function will accept a character as parameter and print a line by using that character. This function does not return any value to main function. The power function will accepts base and power values as parameters and will calculate x^y value and return to the main function for print it.

7.4 STORAGE CLASSES :

All the variables should have a data type and a storage class. To define a variable in C, we need to mention its data type and its storage class. If we do not specify the storage class of a variable in its declaration, the compiler will assume a storage class dependent on the context in which the variable is used. Thus C has got certain default storage classes.

The variables may also be categorized, depending on the place of their declaration, as INTERNAL (local) or EXTERNAL (global). Internal variables are within a particular function, while external variables are declared outside of all functions.

From C compiler point of view, a variable name identifies some physical location within the computer where the string of bits representing the variable's value is stored. There are basically two kinds of locations in a computer where such a value may be kept: memory and CPU register. The variable's storage class determines which of these two locations the value is stored. Moreover, a variable's storage class tells us:

a) Where the variable would be stored.
b) What will be the initial value of the variable, if the initial value is not specifically assigned. (i.e. the default initial value)
c) What is the scope of the variable; i.e. in which function the value of the variable would be available.
d) What is the life of the variable; i.e. how long would the variable exist.

Types of Storage Classes :

i. Automatic storage class
ii. Register storage class
iii. Static storage class
iv. External storage class

i) Automatic Variables : Automatic variables are declared inside a function. These are created when the function is called and destroyed automatically when the function is exited. Automatic variables are private (or local) to the function in which they are declared. Because of this property, automatic variables are also referred to as local or internal variables. We may also use the keyword **auto** to declare automatic variables explicitly. One of the important feature of automatic variables is that their value cannot be changed accidentally by what happens in some other functions in the program. This assures that we may declare and use the same name variable in different functions in the same program without causing any confusion to the compiler. The automatic variable declaration may be :

```
main()                              main()
{                                   {
    int n;                              auto int n;
    ---------       (or)                ---------
    ---------                           ---------
}                                   }
```

Program 7.4 :

```
/* PROGRAM TO ILLUSTRATION OF WORKING OF AUTO VARIABLES */
main()
{
    int a=10000;
    fun2();
    printf("%d\n",a);
}
fun1()
{
    int a=100;
    printf("%d\n",a);
}
fun2()
{   int a=1000;
    fun1();
```

```
        printf("%d\n",a);
}
```

OUTPUT:

```
10
100
1000
```

Explanation : In the main function the variable **a** is declared as local and holding the value 10000 this is the last line output. The main function is calling fun2, in this fun2 the variable **a** is declared as local and holding the value of 1000, this is the second line output. This fun2 is calling fun1, in the fun 1 the variable **a** is declared as local and holding the value 100, this is the first line output.

ii) External Variables : Some variables are alive and active throughout the entire program are known as external variables or global variables. External variables are declared outside of all function in the program. The global variables can be accessed by any function in the program. Once a variable has been declared as global, any function can use it. and change its value. Then subsequent function can refer only that new values.

Program 7.5 :

```
/*  PROGRAM TO ILLUSTRATE PROPERTIES OF GLOBAL VARIABLES */
int a;
main()
{
        a=25;
        printf("a  =  %d\n",a);
        printf("a  =  %d\n",function1());
        printf("a  =  %d\n",function2());
        printf("a  =  %d\n",function3());
}
function1()
{
        a=a+10;
        return(a);
}
function2()
{
        int a;
        a=10;
        return(a);
}
function3()
{
        a=a+10;
        return(a);
}
```

OUTPUT:

```
a  =  25
a  =  35
a  =  10
a  =  45
```

Explanation : In the above program the variable **a** is declared as global. In the main function 25 is assigned to this variable **a**. The first line of output prints this value i.e. 25. In the function1 10 is added to the variable **a** now its value is 35. This is the second line output. In the function3 10 is added to the variable **a** now its value becomes 45, this is the last line output. In the function2 the variable **a** declared as local is holdin value 10, this is the third line output.

iii) Static Variables : A variable can be declared as static by using **static** keyword. The value of static variables persists until the end of the program. A static variables may be either an internal type or an external type, depending on the place of the declaration. Internal static variables are those which are declared inside a function. The scope of internal static variable extend up to the end of the function. Therefore internal static variables are similar to auto variables, except that they remain in existence (alive) throughout the program. A static variable is initialized only once, when the program is compiled; it is never initialized again.

Program 7.6 :

```
/*  PROGRAM TO ILLUSTRATE PROPERTIES OF STATIC VARIABLES  */
main()
{
        int c;
        for(c=1;c<=3;c++)
        fun();
}
fun()
{
        static int a=5;
        a=a+3;
        printf("a  =  %d\n",a);
}
```

OUTPUT:

```
a  =  8
a  =  11
a  =  14
```

Explanation : In the above program the first call of fun, **a** is initialized with 5 and add 3 to it, now the new value is 8. Because **a** is static, this value persists and therefore, the next call adds another 3 to **a** now the new value is 11. The value of **a** becomes 14 during te third call of fun.

iv) Register Variables : Some of the variables can store their values in one of the machine's registers, instead of keeping in the memory (where normal variables are stored). Since a register access is much faster than a memory access. Keeping the frequently accessed variables (like loop control variables) in the register will lead to faster execution. It can be done :

```
register int i;
```

Most compilers allow only int or char variables to be placed in the register, since only a few variables can be placed in the register. However C will automatically convert register variables into non-register variables once the limit is reached.

7.5 SCOPE RULES :

The scope of a variable is the portion of a program in which the variable may be visible or available. A variable is said to have ***Block scope*** if it is recognized only within the block. A variable with block scope is known as a ***Local Variable*** or ***Private variable or Internal Variable.***

If the declaration of a variable appears outside of all the blocks in a program, the variable is said to have ***File Scope.*** Such a variable can be recognized in the entire program. A variable with the file scope is known as a ***global variable*** or ***public variable or external variable.*** The same variable name may appear in different scopes. It is the function of the compiler to decide whether the different occurrences of the variable refer to the same variable or not. Based on the scope of the variable, the compiler decides the linkage of the different occurrences of the same variable.

7.6 STANDARD LIBRARY FUNCTIONS :

The C language supports a number of library functions that carry out various commonly used operations or calculations. A library function is access by writing the function name followed by a list of arguments.

Commonly used Library Functions :

Function	Type	Purpose
abs(x)	int	Returns the absolute value of x
cos(d)	double	Return the cosine of d
exp(d)	double	Raise e to the power d (e=2.71828)
log(d)	double	Return the natural logarithm of d
log10(d)	double	Return the logarithm (base 10) of d
pow(d1, d2)	double	Return d1 raised to the d2 power
sin(d)	double	Return the sine of d
sqrt(d)	double	Return the square root of d
tan(d)	double	Return the tangent of d
toascii(c)	int	Convert value of argument to ASCII
tolower(c)	int	Convert letter to lowercase

toupper(c)	int	Convert letter to uppercase
isalpha(c)	int	Determine if argument is alphabetic, Returns nonzero value if true, 0 otherwise
isdigit(c)	int	Determine if argument is a decimal digit, Returns nonzero value if true, 0 otherwise
islower(c)	int	Determine if argument is lowercase, Returns nonzero value if true, 0 otherwise
isupper(c)	int	Determine if argument is uppercase, Returns nonzero value if true, 0 otherwise
isascii(c)	int	Determine if argument is an ASCII character, Returns nonzero value if true, 0 otherwise

7.7 RECURSIVE FUNCTION :

The function called by itself is called recursive function and this process is often referred to as recursion

Ex:-

```
main()
{
    printf("Welcome to SHREETECH \n");
    main();
}
```

Important Conditions : There are two important conditions that must be satisfied by any recursive procedure.

1. Each time a procedure calls itself, it must be nearer to a solution.
2. There must be a decision criterion for stopping the computation.

Types of recursion : There are two types of recursions.

1. The first type concerns *recursively defined functions* (or *primitive recursive functions*). Factorial function is an example of this kind.
2. The second type of recursion is the *recursive use of a procedure* (*nonprimitive recursive*). Ackermann's function is an example of this kind.

Factorial of a Given Number :

$$fact(n) = \begin{cases} 1, & \text{if } n = 0 \\ n * fact(n - 1), & \text{otherwise} \end{cases}$$

Here fact(n) is defined in terms of fact(n-1), which in turn is defined in terms of fact(n-2), etc., until fact(0) is reached, whose value is given as "one".

Program 7.7 :

```
/*  FACTORIAL OF A GIVEN NUMBER BY USING RECURSION  */
#include<stdio.h>
void main()
{
    int n, rst;
    int fact(int);
    printf(" Enter any number : ");
    scanf("%d",&n);
    rst=fact(n);
    printf(" Factorial of %d is : %d\n",n,rst);
}
int fact(int x)
{
    if(x==0)
        return 1;
    else
        return(x*fact(x-1));
}
```

OUTPUT:

Enter any number : 5
Factorial of 5 is : 120

Explanation : In the above program accepted value n will be passed to the fact function. The fact function will call itself till the argument becomes 0. The final value will be saved in rst variable.

Fibonacci Number :

$$fib(n) = \begin{cases} 1, & \text{if } n = 0 \\ 1, & \text{if } n = 1 \\ Fib(n-1)+fib(n-2), & \text{otherwise} \end{cases}$$

Here fib(0) is 1 and fib(1) is also 1and fib(n) is defined in terms of fib(n-1)+fib(n-2), like :

fib(0) = 1
fib(1) = 1
fib(2) = fib(1)+fib(0)
fib(3) = fib(2)+fib(1)
fib(4) = fib(3)+fib(2)

Program 7.8 :

```
/*      CALCULATED THE SPECIFIED NUMBER FIBONACCI NUMBER BY USING RECURSION */
#include<stdio.h>
void main()
{
      int n, rst;
      int fibo(int);
      printf(" Enter any number : ");
      scanf("%d",&n);
      rst=fibo(n);
      printf(" the %dth Fibonacci number is : %d\n",n,rst);
      getch();
}
int fibo(int x)
{
      if(x==0||x==1)
          return 1;
      else
          return(fibo(x-1)+fibo(x-2));
}
```

OUTPUT:

Enter any number : 8
The 5th Fibonacci number is : 13

Explanation : In the above program accepted value n will be passed to the fibo function. The fibo function will call itself till the argument becomes 0 or 1. The final value will be specified Fibonacci number.

GCD of two number :

$$gcd(a,b) = \begin{cases} a, & \text{if } b = 0 \\ gcd(b, a\%b), & \text{otherwise} \end{cases}$$

Here gcd(a,b) is defined in terms of gcd(b, a%b), The termination point is b=0.

Program 7.9 :

```
/*  Calculating the GCD of given two numbers by using recursion  */
#include<stdio.h>
void main()
{
      int a, b, rst;
      int gcd(int,int);
      clrscr();
      printf(" Enter first number : ");
      scanf("%d",&a);
```

```
        printf(" Enter second number : ");
        scanf("%d",&b);
        rst=gcd(a,b);
        printf(" GCD of %d  and  %d  is :  %d \n",a,b,rst);
        getch();
}
int gcd(int x, int y)
{
        if(y==0)
                return x;
        else
                return(gcd(y,x%y));
}
```

OUTPUT:

Enter first number : 18
Enter second number : 45
GCD of 18 and 45 is : 9

Explanation : In the above program accepted two integer values a and b will be passed to the gcd function. The gcd function will call itself till the second argument becomes 0. The final value will be saved in rst variable.

7.8 HEADER FILES :

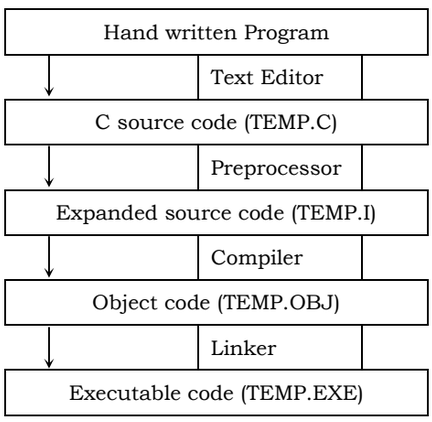
C compiler provides a number of built-in or library functions. Certain definitions and declarations are required when these library functions are to be used. They are available in a file called ***Header File.*** Each header file contains the library functions along with the necessary definitions and declarations. Standard I/O library functions are available in a file named ***stdio.h*** (standard input output header file). The following line is included in a C program whenever standard I/O functions are used.

#include<stdio.h>

It causes the contents of the header file to be included within the program. If it is not included in a program, the user must define and declare the necessary I/O functions.

7.9 C PREPROCESSOR :

The C Preprocessor is a program that process is our program before it is passed to the compiler. Here if the source code is stored in a file temp.c then the expanded source code gets stored in a file temp.i. When this expanded source code is compiled the object code gets stored in temp.obj. When this object code is linked with the object code of the library function the resultant executable code gets stored in temp.exe.



A preprocessor carries out the following actions like (i) Simple string replacement, (ii) Macro expansion, (iii) File inclusion and (iv) Conditional inclusion, on the source file before it is passed to the compiler.

Simple String Replacement : It is replacement of a string which is accomplished through #define statement. The general form of directive is :

```
        #define macro_name sequence_of_characters
```

Ex :

```
        #define EX “This is my example program \n”
```

When the identifier EX is encountered the compiler will replace it by the string “This is y example program \n”.

Macro expansion : A macro is pseudo function i.e. it appears like a function in its form and usage. Macro can be classified into two groups they are (i) Simple macro definition and (ii) Macro definition with arguments.

Simple Macro Definition : A macro is simply substitution string that is placed in the program.

Ex :

```
#define PRINT printf("Welcome to Macros");
```

When we want to display this message, we just use the instruction PRINT.

Once a macro name has been defined, it may be used as part of the definition of other macro names.

Ex :

```
#define TEN 10
#define HUNDRED (TEN * TEN)
#define THOUSAND (HUNDRED * TEN)
```

Macro definition with arguments : We can define macros with parameters. General form :

```
#define macro_name(n1, n2, . . .nn)
```

Ex :

```
#define SQUARE(x) ((x) * (x))
#define MAX(a,b) a>=b?a:b
```

File Inclusion : Some function from another file is required in the current program, then that file can be included in the current program by **include** statement. Then all function written in the included file can be used in the furrent file. This saves time and reusability to avoid rewriting of readely available function.

Ex :

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

Conditional Inclusion : Conditional inclusion is used for conditional selection of parts of the source file. Conditional selection is rarely performed using #defined values.

#if and #endif : #if evaluates a constant integer expression. #endif is a delimiter of end of statement. General form of #if is :

```
#if constant_expression
        Statements
#endif
```

#else : It works much similar to the else part in the C language.

Ex :

```
#define RUNS 60
…….
…….
#if RUNS>=100
   printf("Century\n");
#else
   printf("Not a Century\n");
#endif
```

#elif : It works much similar to the else-if in the C language.

7.10 OBJECTIVE TYPE QUESTIONS

1. Storage class controls [ ]
   a) life time of a variable b) scope of a variable
   c) linkage of a variable d) all the above
2. Which is not a storage class [ ]
   a) auto b) struct c) typedef d) static
3. Automatic storage class has [ ]
   a)temporary storage b) block scope
   c) persistent storage d)options c&b
4. The register storage class has [ ]
   a)block scope b)persistent storage
   c)temporary storage d) options a&b
5. The storage class type of external static has [ ]
   a)persistent storage b)block scope
   c)file scope d)options a and c

6. The typedef statement is used to [ ]
   a) create a new data type b) rename the existing data type
   c) to define a storage class d) create a structure
7. Which storage class may help in faster execution [ ]
   a) static b) extern c) register d) auto
8. Which storage class specifies local variables: [ ]
   a)auto b)register
   c)internal static d) all the above
9. External variable declaration uses: [ ]
   a) the keyword external b) the keyword extern
   c) no keyword d) the keyword auto
10. What would be the error in the following function at the time of compilation? [ ]

```
f( int x, int y)
{
        int  x;
        x=20;
        return x;
 }
```

   a) missing parentheses in return statement
   b) the function should be defined as intf(int x, int y)
   c) redeclaration of x
   d) None oth the above
11. The following code prints 'Computer' _____ number of times.

```
main()                                                    [ ]  {
    printf("\nComputer");
    main( );
}
```

   a) infinite number of times b) 32767 times
   c) 65536 times d) Till the stack doesn't overflow
12. What is the output of the following program? [ ]

```
main( )
{
     int  i=0;
     while(i<5)
     {
          sum(i);
          i++;
     }
}
void sum(i)
int i;
{
      static int k;
      printf("%d",k++);
      k++;
}
```

   a) 0 1 2 3 4 b) 1 2 3 4 5 c) 1 3 5 7 9 d) 0 2 4 6 8

ANSWERS

1.	d	2.	b	3.	d	4.	d	5.	d
6.	a	7.	c	8.	d	9.	b	10.	c
11.	d	12.	d						

8 – ARRAYS and STRINGS

8.1 INTRODUCTION :

Array is a homogeneous compound data type, or an array is a group of homogeneous data items sharing a common name. The ability to use a single name to represent a collection of items is achieved through arrays. The individual values are called ELEMENTS and the complete set of values are referred as an array. A particular element can be referred by writing a number called INDEX number or SUBSCRIPT in brackets after the array name.

Array properties:

1. The type of an array is the data type of its elements.
2. The location of an array is the location of its first element.
3. The length of an array is the number of data elements in the array.
4. The size of an array is the length of the array times the size of an element.

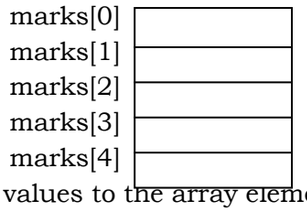
Arrays whose elements are specified by one subscript are called single subscripted or single dimensional array. Analogous array whose elements are specified by two and three subscripts are called two-dimensional or double subscripted and three-dimensional or triple-subscripted arrays respectively.

8.2 ONE-DIMENSIONAL ARRAY :

A list of items can be given one variable name and elements are specified by one subscript are called a single-subscripted variable or a one-dimensional array.

int marks[5];

The computer will reserve five storage locations to store five integer values, as shown below.



The values to the array elements is assigned and stored as follows

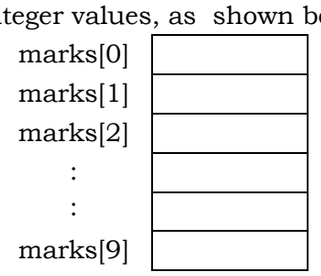
marks[0] = 45
marks[1] = 65
marks[2] = 10
marks[3] = 93
marks[4] = 40

Array declaration : In C arrays must be declared before they are using. The type of the array is of that elements that will hold in the array. Array of character elements are is called a **string**. When we declare an array you must specify (either directly or indirecty) how many elements the array will have i.e. the size of the array. This size of the array should be specified between in a pair square brackets ([]). This size of the array cannot be a variable and has to be an integer constant. The general form of array declaration is

type array_name[size];

Ex: int marks[10]

With the above declaration the computer will reserve 100 storage locations to store 100 integer values, as shown below.



Array Initialization : Elements of an array can be assigned initial values by the following array definition with a list of initializes enclosed in braces and separated by comma.

Ex:- int marks[5] = {65, 98, 62, 48, 57};

Here it defines the array **marks** to contain five integer elements and initializes marks[0] to 65, marks[1] to 98, marks[2] to 62, marks[3] to 48 and marks[4] to 57. The array initialization can be done in the following ways :

1. If the number of initializers is less than the number of elements in the array, the remaining elements are set to zero.

   Ex:- int m[5]={3,4,8}; is equivalent to
   int m[5]={3,4,8,0,0};

2. If initializers have been provided for an array, it is not necessary to explicitly specify the array length, in which case the length is derived from the initializers.
   Ex:- int m[]={3,7,3,9,10}; is equivalent to
   int m[5]={3,7,3,9,10};
3. A character array may be initialized by a string constant, resulting in the first element of the array being set to the first character in the string, the second element to the second character, and so on. The array also receives the terminating '\0' in the string constant.
   Ex:- char name[10]="COMPUTERS"; is equivalent to
   char name[10]={'C','O','M','P','U','T','E','R','S','\0'};

8.3 STORING AND ACCESSING ELEMENTS :

The array elements can be accessed by simply specifying the name of the array followed by an index within the square braces. In the same way we can store the value in the specified indexes. You can enter the element values of the array by using scanf() statement in the 'for' loop.

Ex.

Program 8.1 :

```
// PROGRAM TO READ AND PRINT AN ARRAY
#include<stdio.h>
main()

{
    int m[100], i, n;
    printf("Enter how many elements do you want? :  ");
    scanf("%d",&n);
    for(i=0; i<n; i++)
    {
        printf("Enter the value of m[%d] ;  ",i+1);
        scanf("%d",&m[i]);
    }
   printf("The values of the array a is : \n");
   for(i=0; i<n; i++)
        printf("m[%d] = %d\n",i+1,m[i]);
}
```

OUTPUT :

```
Enter how many elements do you want? :  5
Enter the value of m[1] ;  10
Enter the value of m[2] ;  20
Enter the value of m[3] ;  30
Enter the value of m[4] ;  40
Enter the value of m[5] ;  50
The values of the array a is :
m[1] = 10
m[2] = 20
m[3] = 30
m[4] = 40
m[5] = 50
```

Explanation : In the above program will accept the number of elements are present in the array from the user as n. Accepts array element from the user and print it by using for loops.

Size of an array : By using the sizeof() operator, we can identify the total number of memory used by the array.

```
int marks[10];
```

with the above example the memory occupied by the array marks is

```
sizeof(marks);
```

will returns 20 since 10 ints will occupy 10 * 2 = 20 bytes.

If the entered index of an array exceeds the size leads unpredictable results.

8.4 TWO-DIMENSIONAL ARRAYS :

To store a table of values we will go for ***two-dimensional*** arrays. Array whose elements are specified by two subscripts are called two-dimensional or double

subscripted arrays. It is nothing but collection of one-dimensional arrays, which are placed one after another.

Array declaration: The two-dimensional array can be declared by specifying the type of the array, then the name of the array variable, and then the number of rows and column elements should be specified between a pair square brackets ([][]). Note that rows and column values cannot be a variable and has to be an integer constant.

General format :

```
type array_name[rwo_size][column_size];
```

Ex:-
```
int marks[100][5];
float salary[1000][4];
```

Array Initialization : Elements of an array can be assigned initial values by following the array definition with a list of initializers enclosed in braces and separated by comma.

Ex:- `int marks[2][3] = {65, 98, 62, 48, 57, 40};`

Here it defines the marks array contain 2 rows and 3 columns totally six integer elements and initializes marks[0][0] to 65, marks[0][1] to 98, marks[0][2] to 62, marks[1][0] to 48, marks[1][1] to 57 and marks[1][2] to 40.

8.5 MULTI-DIMENSIONAL ARRAYS :

C program allows array of multi dimensions also. The syntax of multi-dimensional array is :

```
type array_name[size1][size2]. . . .[sizen]
```

8.6 STRINGS :

A string is nothing but single dimensional char array. So whatever rules are applicable to single dimensional arrays are applicable to strings also. Any string constant defined between double quotation marks is a string.

Ex:- "SHREETECH Computers"

Declaration and Initialization : A string declaration is as follows :

```
char string_name[size]
```

The size determines the number of characters in the string.

Ex:- `char city[25]`

The character array may be initialized when they are declared, like :

```
char city[25] = “Hyderabad"
```

When a compiler assigns a character string to a character array, it automatically supplies a NULL character '\0' at the end of the string. Therefore the size should be equal to the maximum number of characters in the string plus one.

Reading a string : The input function scanf() can be used with %s format descriptor to read a string. The symbol & is not required for strings in scanf statement. This scans only upto white space.

Ex:-
```
char city[20];
scanf("%s",city);
```

The function **gets** accepts a character array as parameter. It reads string (until new line) from the standard input. If no input is available (end of file is encountered) it returns a special value called NULL.

Ex:-
```
char city[20];
gets(city);
```

Writing a string : The print function printf() can be used with %s format descriptor to print a string.

Ex:-
```
char city[20];
scanf("%s",city);
printf("%s\n",city);
```

The function **puts** accepts a string as parameter, to print the string on the standard output and further automatically issues a new line.

Ex:-
```
char city[20];
gets(city);
puts(city);
```

8.7 STRING OPERATIONS :

The following are the operations that can be performed on strings.

1. Reading and Writing of Strings
2. Concatenating of strings
3. Copying one string into another
4. Comparing Strings
5. Extracting a portion of a string

6. Converting the string from lower case to uppercase

'C' provides string handling functions to perform the above specified operations. The following are the

8.8 STRING HANDLING FUNCTIONS :

strcpy() : It is used to copy one string into another string.

Syntax :

strcpy(str1, str2)

Here str1 is the source string and str2 is the target string.

strcmp() : It is used to compare two strings character by character and returns –1 (or) 0 (or) 1.

Syntax :

strcmp(str1, str2)

if the ASCII value of the character of the first string is less than the second string it returns –ve. If both strings are equal it returns zero. If the ASCII value of the character of the first string is greater than the second string it returns +ve.

strcat() : It is used to concatenete two strings i.e. it appends one string into another string.

Syntax :

strcat(str1, str2)

Here string str2 is appended to the end of the string str1.

strlen() : It is used to count the number of characters in the string.

Syntax :

strlen(str1)

strlwr() : It is used to convert any upper case letters into its equivalent lower case letters.

Syntax :

strlwr(str1)

strupr() : It is used to convert any lower case letters into its equivalent upper case letter.

Syntax :

strupr(str1)

8.9 OBJECTIVE TYPE QUESTIONS

1. Array is used to represent: [ ]
   a) a list of data items of integer data type
   b) a list of data items of real data type
   c) a list of data items of different data type
   d) a list of data items of same data type
2. One-dimensional array is known as [ ]
   a) a vector b) a table
   c) a matrix d) an array of arrays
3. The array elements are represented by [ ]
   a)index values b)subscripted variables
   c)array name d) size of array
4. Array elements occupy [ ]
   a)subsequent memory locations
   b) random location for each element
   c) varying length of memory locations for each elements
   d) no space in memory
5. Array subscripts in C always start at [ ]
   a) –1 b) 0 c) 1 d) any value
6. Identify the correct declaration: [ ]
   a) int a[10][10] b) int a(10)(10)
   c) int a [10,10] d) int a(10,10)
7. Maximum number of elements in the array declaration int x[5][8]:
   [ ]
   a) 28 b) 34 c) 45 d) 40
8. The value within the [] in an array declaration specifies
   [ ]
   a) subscript value b) size of the array
   c) value of the array element d) array bound

9. When should an array be used? [ ]
   a)When we need to hold variable constants
   b) When we need to hold data of the same type
   c) When we need to obtain automatic memory cleanup functionality
   d) when we need to hold data of different types
10. Which is string related function: [ ]
   a) strcmp() b) strcpy() c) strcat() d) all above
11. The function to compare two strings is: [ ]
   a) strcat() b) strcmp() c) strcpy() d) strlen()
12. The function to copy a string into another [ ]
   a)strcat() b)strcmp() c)strcpy() d)all above
13. When two strings are equal then strcmp() return: [ ]
   a) 1 b) 2 c) –1 d) 0
14. To find length of a string function is [ ]
   a)strlen() b)strcat() c)strlength() d)lengthstr()
15. Program execution starts with [ ]
   a) the function which is first defined b) main() function
   c) the function which is last defined
   d) the function other than main()
16. How many main() functions can be defined in a C program? [ ]
   a) 1 b) 2 c) 3 d) any number of times
17. A function is identified by an open parenthesis following [ ]
   a) a keyword b) an identifier other than keywords
   c) an identifier including keywords d) an operator
18. Parameters are used [ ]
   a) to return values from the called function
   b) to send values from the calling function
   c) options a and b
   d) to specify the data type of the return value
19. The default return data type in function definition is [ ]
   a)void b)int c)float d)char

ANSWERS

1.	d	2.	a	3.	b	4.	a	5.	b
6.	a	7.	d	8.	b	9.	b	10.	d
11	b	12	c	13	d	14	a	15	b
16	a	17	b	18	c	19	b		

8.10 EXAMPLE PROGRAMS

Program 8.2 :

```
/* FINDOUT THE MAXIMUM ELEMENT FROM THE GIVEN ARRAY */
#include<stdio.h>
main()

{
    int a[50],i,n,j,max;
    clrscr();
    printf("Enter the size of the array : ");
    scanf("%d",&n);
    for(i=0;i<n;i++)
    {
        printf("Enter a[%d] = ",i+1);
        scanf("%d",&a[i]);
    }
    max=0;
    for(j=0;j<n;j++)
        if(a[j]>=max)
            max=a[j];
    printf("Maximum value from the given array : %d\n",max);
}
```

OUTPUT:

```
Enter the size of the array : 6
Enter a[1] = 8
Enter a[2] = 2
Enter a[3] = 6
Enter a[4] = 5
Enter a[5] = 1
Enter a[6] = 3
Maximum value from the given array : 8
```

Explanation : In the above program will accept the number of elements are present in the array from the user as n. Accepts array element from the user and print it by using for loops. The max variable is initialized with zero and compare all the array elements, if array element is larger than the max value the max variable holds the array element value. Finally it will print the max value from the array.

Program 8.3 :

```
/* FINDOUT THE NUMBER OF PASS AND FAIL FROM THE GIVEN MARKS ARRAY
*/
#include<stdio.h>
main()
{
        int m[50],i,n,pass=0;
        clrscr();
        printf("Enter the number of students : ");
        scanf("%d",&n);
        for(i=0;i<n;i++)
        {
                printf("Enter  %d student marks  =  ",i+1);
                scanf("%d",&m[i]);
        }
        for(i=0;i<n;i++)
                if(m[i]>=35)
                        pass++;
        printf("Number of passed students are  : %d\n",pass);
        printf("Number of failed students are  : %d\n",n-pass);
}
```

OUTPUT:

```
Enter the number of students :  8
Enter 1 student marks = 57
Enter 2 student marks = 23
Enter 3 student marks = 79
Enter 4 student marks = 82
Enter 5 student marks = 34
Enter 6 student marks = 51
Enter 7 student marks = 06
Enter 8 student marks = 65
Number of passed students are : 5
Number of failed students are : 3
```

Explanation : The above program will accept the number of students are present in the array from the user as n. Accepts individual student marks from the user by using for loop. The pass variable is initialized with zero. All the array elements will check that is it gereater than or equal to 35, if so pass will be incremented by 1. It will be the no. of students pass and no. students fail is n-pass.

Program 8.4 :

```
/* ADD 5 GRACE MARKS FOR FAILED CANDIDATES FROM THE GIVEN ARRAY
*/
#include<stdio.h>
main()
{
```

```
        int m[50],i,n;
        clrscr();
        printf("Enter the number of students : ");
        scanf("%d",&n);
        for(i=0;i<n;i++)
        {
                printf("Enter  %d student marks  =  ",i+1);
                scanf("%d",&m[i]);
        }
        for(i=0;i<n;i++)
                if(m[i]<35)
                        if(m[i]+5>35)
                                m[i]=35;
                        else
                                m[i]=m[i]+5;
        printf("The modified marks are :  \n");
        for(i=0;i<n;i++)
                printf("\t m[%d]  =  %d\n",i+1,m[i]);
}
```

OUTPUT:

```
Enter the number of students :  8
Enter 1 student marks = 57
Enter 2 student marks = 23
Enter 3 student marks = 79
Enter 4 student marks = 82
Enter 5 student marks = 34
Enter 6 student marks = 51
Enter 7 student marks = 32
Enter 8 student marks = 29
The modified marks are :
        m[1] = 57
        m[2] = 28
        m[3] = 79
        m[4] = 82
        m[5] = 35
        m[6] = 51
        m[7] = 35
        m[8] = 34
```

Explanation : The above program will accept the number of students are present in the array from the user as n. Accepts individual student marks from the user by using for loop. The pass variable is initialized with zero. All the array elements will check that is it less than 35 add five grace marks, after adding grace marks if it is greater than 35 it is rounded to 35, the modified value will be stored into same location.

Program 8.5 :

```
/*  TRANSPOSE OF THE GIVEN MATRIX  */
#include<stdio.h>
main()
{
        int a[5][5];
        int i,j,row,col;
        clrscr();
        /*  Collecting the number of rows and number of columns   */
        printf("Enter how many rows are there : ");
        scanf("%d",&row);
        printf("Enter how many columns are there : ");
        scanf("%d",&col);
        /* Matrix Reading   */
        for(i=0;i<row;i++)
        {
                for(j=0;j<col;j++)
                {
```

```
            printf("Enter a[%d][%d] = ",i+1,j+1);
            scanf("%d",&a[i][j]);
                }
        }
        printf("  The transposed matrix is  :  \n");
        /*   Printing the transpose matrix   */
        for(i=0;i<row;i++)
        {
                for(j=0;j<col;j++)
                        printf("\t%d",a[j][i]);
                printf("\n");
        }
}
```

```
Enter how many rows are there : 3
Enter how many columns are there : 3
Enter a[1][1] = 2
Enter a[1][2] = 4
Enter a[1][3] = 6
Enter a[2][1] = 3
Enter a[2][2] = 5
Enter a[2][3] = 7
Enter a[3][1] = 9
Enter a[3][2] = 0
Enter a[3][3] = 8
The transposed matrix is  :
        2       3       9
        4       5       0
        6       7       8
```

Explanation : Number of Rows and Columns will be accepted by user and get the matrix elements from the user. By using nester for loop the transposed matrix can be printed.

Program 8.6 :

```
/*  ADDITION OF TWO MATRICES   */
#include<stdio.h>
main()

{
        int a[5][5], b[5][5], c[5][5];
        int i,j,row,col;
        clrscr();
        /*  Collecting the number of rows and number of columns   */
        printf("Enter how many rows are there : ");
        scanf("%d",&row);
        printf("Enter how many columns are there : ");
        scanf("%d",&col);
        /*   First Matrix Reading   */
        for(i=0;i<row;i++)

                for(j=0;j<col;j++)
                {
                        printf("Enter a[%d][%d] = ",i+1,j+1);
                        scanf("%d",&a[i][j]);
                }
        }
        /*  Second  Matrix Reading   */
        for(i=0;i<row;i++)
        {
                for(j=0;j<col;j++)
                {
                        printf("Enter b[%d][%d] = ",i+1,j+1);
```

```
                scanf("%d",&b[i][j]);
            }
        }
        /*  Addition of First and Second Matrices   */
        for(i=0;i<row;i++)
            for(j=0;j<col;j++)
                c[i][j]=a[i][j]+b[i][j];
        /*  Printing the resultant matrix   */
        for(i=0;i<row;i++)
        {
            for(j=0;j<col;j++)
                printf("  %2d",a[i][j]);
            printf("\t\t");
            for(j=0;j<col;j++)
                printf("  %2d",b[i][j]);
            printf("\t\t");
            for(j=0;j<col;j++)
                printf("  %2d",c[i][j]);
            printf("\n");
        }
}
```

Explanation : Number of Rows and Columns of two matrices will be accepted by user and get the two matrices elements from the user. By using nested for loop the addition of two matrices can be performed and stored into other matrix. Finally three matrices can be printed.

Program 8.7 :

```
/*  FIND OUT THE LENGTH OF THE GIVEN STRINGS   */
#include<stdio.h>
main()

{
        char str[80];
        int i=0,len;
        clrscr();
        printf("Enter a string : ");
        gets(str);
        while(str[i]!=NULL)
            i++;
        printf("  Length of the given string is : %d\n",i);
}
```

OUTPUT:

Enter a string : Computer

Length of the given string is : 8

Explanation : The above program accepts a string from user. The variable i is initialized with zero. If ith character of the string is not NULL i wil be incremented by 1. This process will repeat till reacing the NULL haracter. Finally i is holding the length of the given string.

Program 8.8 :

```
/* CONVERT THE GIVEN STRING INTO LOWER CASE LETTERS    */
#include<stdio.h>
main()

{
        char str[80], lwr[80];
        int i=0;
        clrscr();
        printf("Enter a string : ");
        gets(str);
        while(str[i]!=NULL)
        {
                lwr[i]=str[i]+32;
```

```
            else
                lwr[i]=str[i];
            i++;
        }
        lwr[i]='\0';
        printf("  The original string is :  ");
        puts(str);
        printf("  Lowercase equivalent of the given string is :  ");
        puts(lwr);
}
```

OUTPUT:

Enter a string : CompuTER
Lowercase equivalent of the given string is : computer

Explanation : The above program accepts a string from user. The variable i is initialized with zero. If ith character of the string is inbetween A to Z, 32 will be added to it and will be stored into lwr string otherwise it will be stored same. Finally the lwr string will be printed.

Program 8.9 :

```
/* FIND OUT THE NUMBER OF LETTERS, WORDS IN A GIVEN STRING  */
#include<stdio.h>
main()
{
        char str[80];
        int i=0, wc=1;
        clrscr();
        printf("Enter a string : ");
        gets(str);
        while(str[i]!=NULL)
        {
            if(str[i]==' '&&str[i+1]!=' ')
                wc++;
            i++;
        }
        printf("  The Character count is :  %d\n",i);
        printf("  The word count is : %d\n",wc);
}
```

OUTPUT:

Enter a string : Computer Science and Engineering
The Character count is : 32
The word count is : 4

Explanation : The above program accepts a string from user. The variables i and wc are initialized with zero. Till getting the NULL character in the string the variable i will be incremented by 1. If ith character is blank space and i+1th character is not blank space wc will be incremented by 1. Now i value will be the character count and wc will be the word count.

Program 8.10 :

```
/* TO CONCATNATE THE GIVEN TWO STRINGS INTO THIRD STRING  */
#include<stdio.h>
main()
{
        char s1[80], s2[80], s3[160];
        int i=0,j=0;
        clrscr();
        printf("Enter the first string : ");
        gets(s1);
        printf("Enter the second string : ");
        gets(s2);
        while(s1[i]!=NULL)
        {
            s3[i]=s1[i];
            i++;
        }
```

```
        while(s2[j]!=NULL)
        {
                s3[i]=s2[j];
                i++;
                j++;
        }
        s3[i]='\0';
        printf("  The first string is :  ");
        puts(s1);
        printf("  The second string is :  ");
        puts(s2);
        printf(" The concatenated string is  : ");
        puts(s3);
}
```

OUTPUT:

Enter the first string : Software
Enter the second string : Engineer
The first string is : Software
The second string is : Engineer
The concatenated string is SoftwareEngineer

Explanation : The above program accepts two strings as s1 and s2 from user. Till getting the NULL character of the first string all the characters will be copied into s3 string. Immediately second string characters also will copied. Now s3 will be concatenated of first and second strings.

Program 8.11 :

```
/*   FIND OUT THE NUMBER OF VOWELS ARE IN A GIVEN STRING  */
#include<stdio.h>
main()
{
        char str[80];
        int i=0,vc=0;
        clrscr();
        printf("Enter a string : ");
        gets(str);
        while(str[i]!=NULL)
        {
           if(str[i]=='a'||str[i]=='e'||str[i]=='i'||str[i]=='o'||str[i]=='u'||
                 str[i]=='A'||str[i]=='E'||str[i]=='I'||str[i]=='O'||str[i]=='U')
              vc++;
           i++;
        }
        printf("  The number of vowels are in the given string is  :  %d\n",vc);
}
```

OUTPUT:

Enter a string : Computers
The number of vowels are in the given string is : 3

Explanation : The above program accepts a string from user. The variables i and vc are initialized with zero. Till getting the NULL character in the string the variable i will be incremented by 1. Each ith character can be checked whether is it vowel or not, if so vc will be incremented by 1.

Program 8.12 :

```
/*  CONVERT THE GIVEN STRING INTO UPPER CASE LETTERS    */
#include<stdio.h>
main()
{
        char str[80], upr[80];
        int i=0;
        clrscr();
        printf("Enter a string : ");
        gets(str);
        while(str[i]!=NULL)
```

```
    {
        if(str[i]>='a'&&str[i]<='z')
            upr[i]=str[i]-32;
        else
            upr[i]=str[i];
        i++;
    }
    upr[i]='\0';
    printf("  The original string is :  ");
    puts(str);
    printf("  Upper case equivalent of the given string is :  ");
    puts(upr);
}
```

OUTPUT :

Enter a string : VighNEsh
The original string is : VighNEsh
Upper case equivalent of the given string is : VIGHNESH

Program 8.13 :

/* A Maruthi Car dealer maintains a record of sales of various vehicles in the following form :

Vehicle type	Month of sales	Price (Rs)
Maruthi-800	02/87	75,000
Maruthi-DX	07/87	95,000
Gypsy	04/87	1,10,000
Mruthi Van	08/88	85,000

Write a C program to read this data into a table of strings and output the details of a particular vehicle sold during a specified period. The program should request the user to input the vehicle type and the period (starting month & ending month). */

```
#include<stdio.h>
#include<conio.h>
#include<math.h>
/*  Main program begins  */
main()
{
    int vt[100], date[100], price[100], num=0;
    int cho, x, ivt, ismon, isyear, iemon, ieyear, flag;
    char ovt[4][20]={"Maruthi-800","Maruthi-DX","Maruthi-
            Gypsy","Maruthi Van"}
    do
    {
        clrscr();
        /*  User Menu Code  */
        printf("\n 1 - Create Records ");
        printf("\n 2 - Read Records for a particular vehicle ");
        printf("\n 3 - Exit ");
        printf("\n Enter your choice : ");
        scanf("%d",&cho);
        /* Evaluation of user choice  */
        if(cho==2&&num==0)
        {
            printf("\n\n Records not created yet, Press
                    any key to create records");
        cho=1;
        }
        if(cho>=3&&cho<1)
        {
            printf(" Terminate the program, Press any key ");
```

```
        getch();
        exit();
}
printf("\n   1  -   Maruthi-800");
printf("\n   2  -   Maruthi-Dx");
printf("\n   3  -   Maruthi-Gypsy");
printf("\n   4  -   Maruthi-van");
printf("\n  Enter your choice  :  ");
scanf("%d",&cho);
swich(cho)
{
        case 1:          /*   Addition of records    */
                printf(" Enter no of vehicles  :  ");
                scanf("%d",&num);
                for(x=0;x<num;x++)
                {
                printf("\n  Enter vehicle type    ");
                scanf("%d",&vt[x]);
                if(vt[x]>4||vt[x]<1)
                {
                        printf(" Invalid vehicle type");
                        x--;
                        continue;
                }
                printf(" Enter month of sale  :  ");
                scanf("%d",&date[x][1]);
                printf (" \n\t Enter year of Sale  :  " );
                scanf("%d",&date[x][2]);
                printf("\n\t Enter Price of Vehicle ");
                scanf("%d",&price[x]);
                }
                printf("\n Records created successfully, Press
                        any key to continue");
                getch();
                break;
    case 2:     /*    Display of information */
                printf("\n\t Enter vehicle type  :  ");
                scanf("%d",&ivt);
                printf("\n\t Enter Period of Sales  :  ");
                printf("\n\t Enter Starting Month  :  ");
                scanf("%d",&ismon);
                printf ("\n\t Enter Starting Year  :  ");
                scanf("%d',&isyear);
                printf ("\n\t Enter Ending Month  :  ");
                scanf("%d",&iemon);
                printf ("\n\t Enter Ending Year  :  ");
                scanf("%d",&ieyear);
                if(ivt>4||ivt<l)
                {
                        printf("\n\t Invalid Vehicle Type ");
                        getch();
                        break;
                }
                if(isyear==ieyear)
                {
                       if(ismon>iemon)
                             prihtf("\n\t  Invalid period Format");
                        getch();
                        break;
        }
```

```
                }
                if(isyear>ieyear)
                {
                        printf("\n\t Invalid Period Format");
                        getch();
                        break;
                }
                flag=0;
                printf("\n\t Record for Sales of %s for the period
                        %d / %d to %d / %d \nl\n", ovt[ivt-1], ismon,
                                isyear, iemon, ieyear); for(x=0;x<num;x++)
                {
                        if(ivt==vt[x])
                        {
                        if(date[x][2]>=isyear&&date[x][2]<=ieyear)
                                {
                                   if(date[x][l]>=ismon&&date[x][1]<=ismon)
                                   {
                                       flag=1;
                                       printf("\n\t\t Date of sale %d/%d,
                                               price%d",date[x][1],
                                                       date[x][2],price[x]);
                                   }
                        }
                }
            }
            if(flag==0)
            {
                printf("\n\t No Data available, Vehicle
                        during the specified Period") ;
                }
                getch();
                break;
            }
      }while(cho!=3);
}
```

OUTPUT:

```
enter 1 to create records
enter 2  to read recordsfor a particular vehicle
enter 3 to exit
enter choice 1
Enter 1 for Maruthi-800
Enter 1 for Maruthi-Dx
Enter 1 for Maruthi-Gypsy
Enter 1 for Maruthi-van
Enter total no of vehicles 1
Enter vehicle type  1
Enter month of sale 2
Enter year of sale  87
Enter price of vehicle 75,000
Records successfully created,
press any key to continue
```

Program 8.14 :

```
/*   TO EXTRACT A PORTION OF THE STRING FROM THE GIVEN STRING   */
#include<stdio.h>
main()
```

```
{
        char str[80], ext[80];
        int i=0,j,pos,n;
        clrscr();
        printf("Enter main string : ");
        gets(str);
        printf(" Enter the starting position the string to be extracted:  ");
        scanf("%d",&pos);
        printf(" Enter number of  character to be extracted  :  ");
        scanf("%d",&n);
        /*   Extracting    */
        for(j=pos-1,i=0;i<n;j++,i++)
                ext[i]=str[j];
        ext[i]='\0';
        /*   Print the extracted string */
        printf("  The extracted string is  :  ");
        puts(ext);
        getch();
}
```

OUTPUT:

Enter the main string : SHREETECH Computers
Enter the starting position the string to be extracted : 6
Enter number of character to be extracted : 4
The extracted String is : TECH

====

9 – POINTERS

9.1 INTRODUCTION:

A Pointer is a derived data type in C. It is built from one of the fundamental data types available in C. Pointers contain memory addresses as their values. Pointers are one of the most distinct and exciting features of C language. It has added power and flexibility to the language. Pointers are used frequently in C.

Advantages of pointers :

- Pointers are more efficient in handling arrays and data tables.
- Pointers can be used to return multiple values from a function via function arguments
- Pointers permit references to functions and there by facilitating passing of functions as arguments to other functions.
- The use of pointer arrays to character string results in saving of data storage space in memory.
- Pointers allow C to support dynamic memory management.
- Pointers provide an efficient tool for manipulating dynamic data structures such as structures, linked lists, queues, stacks and trees.
- Pointers reduce length and complexity of programs.
- Pointers increase the execution speed and thus reduce the program execution time.

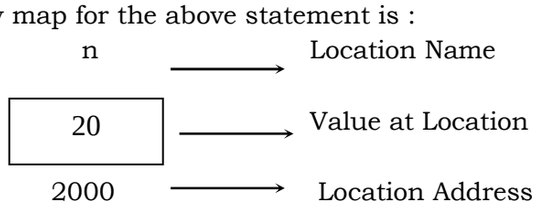
Pointer Declaration : The general syntax of the pointer declaration is :

```
data_type *pointer_name;
```

Here the * tells that variable ***pointer_name*** is pointer type variable. i.e. it holds the address of a variable of type ***data_type***.

For example the statement **int n=20** tells the C compiler to: (i) Reserve space in memory to hold an integer value, (ii) Associate the name n with this memory location and (iii) Store the value 20 at this location.

The memory map for the above statement is :



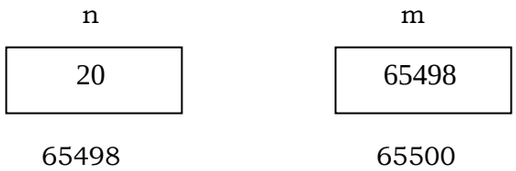
9.2 POINTER OPERATORS :

A pointer operator are an asterisk (*) value at address operator, and an ampersand (&) address operator.

Address of operator : The address of' operator (&) returns the address of a variable. The operand may be a variable or a constant.

Ex : m = &n;

Here m is not an ordinary variable like any other integer variable. It is a variable which contains the address of another variable n. The following memory map would illustrate the content of n and m.



As you can see n's value is 20 and m's value is n's address. Here we can't use m in a program without declaring it. And since m is a variable which contains the address of n, it is declared as

```
int *m;
```

This declaration tells the compiler that m will be used to store the address of an integer value. In other words m points to an integer.

Value at address operator : The value at address operator (*) returns the value stored at a particular address. The ***'value at address'*** operator is also called ***'indirection'*** operator or ***'deference'*** operator.

Ex : x = *m;

Here m is holding the addres of n, and x is holding the value at address of m i.e. n.

Program 9.1 :

```
/* PROGRAM TO DEMONSTRATE & AND * OPERATORS    */
```

```
#include<stdio.h>
main()
{
        int n=20;
        printf("  Address of n is : %u\n",&n);
        printf("  Value of n is : %d\n",n);
        printf("  Value of n is : %d",*(&n));
}
```

OUTPUT :

Address of n is : 65500
Value of n is : 20
Value of n is : 20

Explanation : The first printf statement print the address of a variable n, which in this it is 65500. The second printf statement prints the n value. The third printf statement also prints the n value i.e. value at address of 65500. Address of n (&n) is 65500 in this case. The final output of the program says te value of *(&n) is same as n.

Program 9.2 :

```
/* PROGRAM TO PRINT ADDRESS AND THE VALUE OF A VARIABLE  BY  USING
    & AND * OPERATORS    */
#include<stdio.h>
main()
{
        int n=20;
        int *m;
        m=&n;
        clrscr();
        printf("  Address of n is : %u\n",&n);
        printf("  Address of n is : %u\n",m);
        printf("  Address of m is : %u\n",&m);
        printf("  Value of m is : %u\n",m);
        printf("  Value of n is : %d\n",n);
        printf("  Value of n is : %d\n",*(&n));
        printf("  Value of n is : %d",*m);
}
```

OUTPUT:

Address of n is : 65498
Address of n is : 65498
Address of m is : 65500
Value of m is : 65498
Value of n is : 20
Value of n is : 20
Value of n is : 20

Explanation : Here n is declared as ordinary integer variable and m is declared as an integer pointer variable, holding the address of a variable n. The first three printf statements will print the addresses of n and m variables. The next three printf statements will print the value of n by using *, & and combination of these two operators.

9.3 POINTER INITIALIZATION :

A pointer variable may be declared in the either global variable section or local variable section. Global pointer variables gets initialized to a NULL pointer, pointer which contains the address 0. The local pointer variables are not initialized a specific address, so before you use a pointer variable to access, make the pointer points to a valid object.

9.4 POINTER/ADDRESS ARITHMETIC :

Pointer is a data type in C, certain operations are allowed on it. These operations are call **pointer arithemetic** or **address arithemetic**.

Pointer operations :

1. A pointer can be assigned to another. The effect of this operation is to make both the pointers point to the same object.

2. A pointer can be incremented/decremented, i.e. pointer to point to the subsequent/previous elements.
3. Integers can be added/substracted with pointers, i.e. pointer moves forward or backword directions.
4. Two pointers can be subtracted from one anotherng provided that they are pointing to the same array. It will returns the number of elements between them.

Invalid operations :

1. Adding, multiplying and dividing two pointers.
2. Shifting or masking pointers
3. Addition of float or double to pointers
4. Assignment of a pointer of one type to a pointer of another type.

Program 9.3 :

```
/*   PROGRAM TO PERFORM POINTER ARITHEMATIC    */
#include<stdio.h>
main()
{
        int i=5,*i1;
        float j=5.8,*j1;
        char k='z',*k1;
        printf("  Value of i = %d\n",i);
        printf("  Value of j = %f\n",j);
        printf("  Value of k = %c\n",k);
        i1=&i;
        j1=&j;
        k1=&k;
        printf("  The Original value of i1 = %u\n",i1);
        printf("  The Original value of j1 = %u\n",j1);
        printf("  The Original value of k1 = %u\n",k1);
        i1++;
        j1++;
        k1++;
        printf("  New value in i1 = %u\n",i1);
        printf("  New value in j1 = %u\n",j1);
        printf("  New value in k1 = %u\n",k1);
}
```

OUTPUT:

```
        Value of i = 5
        Value of j = 5.800000
        Value of k = z
        The Original value of i1 = 65490
        The Original value of j1 = 65492
        The Original value of k1 = 65497
        New Value in i1 = 65492
        New Value in j1 = 65496
        New Value in k1 = 65498
```

Explanation : The first three printf statements will print the values of i, j and k variables. Second three printf statements will print the values of i1, j1 and k1 variables, i.e. addresses of i, j and k variables. The last three printf statements will print the incremented values of i1, j1 and k1. Here i1 is integer pointer variable because it will incremented by two bytes, j1 is float pointer variable beause it will incremented by four bytes and k1 is character pointer variable because it will incremented by one byte.

9.5 POINTERS AND FUNCTION ARGUMENTS :

Passing address to functions : Arguments can be passed to functions in two ways:

1. Sending the values of the arguments
2. Sending the address of the arguments

Sending the values of the arguments (Call by Value) : In this method the value of each actual argument in the calling function is copied into corresponding formal arguments of the called function. With this method changes made in the formal

arguments in the called function have no effect on the values of the actual arguments in the calling function.

Program 9.4 :

```
/*    PROGRAM TO ILLUSTRATE THE "CALL BY VALUE"    */
#include<stdio.h>
main()
{
     int x,y;
     printf("Enter first value i.e x : ");
     scanf("%d",&x);
     printf("Enter second value i.e y : ");
     scanf("%d",&y);
     swap(x,y);
     printf(" In the main program : \n");
     printf(" x = %d\n",x);
     printf(" y = %d\n",y);
}
swap(int a, int b)
{
     int t;
     printf(" In the swap function : \n");
     printf(" x   =   a   =   %d\n",a);
     printf(" y   =   b   =   %d\n",b);
     t=a;
     a=b;
     b=t;
     printf(" After interchanging : \n");
     printf(" x   =   a   = %d\n",a);
     printf(" y   =   b   = %d\n",b);
}
```

OUTPUT :

```
Enter first value i.e.  x : 43
Enter second value i.e. y : 94
In the swap function :
x   =   a   =   43
y   =   b   =   94
After interchanging :
x   =   a   =   94
y   =   b   =   43
In the main program :
x = 43
y = 94
```

Explanation : Here actual argument in the calling function is copied into corresponding formal arguments i.e. x value will be copied into a variable, y value will be copied into b variable. The swap function will print these a, b values before swapping and after swapping. Within the swap function a and b values are swapped but it will not affect the x and y which are in the main function. Hence x and y values remains same.

Sending the address of the arguments (Call by reference) : In this method the address of actual arguments in the calling function are copied into formal arguments of the called function. This means that using the formal arguments in the called function we can make changes in the actual arguments of the calling function.

Program 9.5 :

```
/*    PROGRAM ILLUSTRATE THE "CALL BY REFERENCE"    */
#include<stdio.h>
main()
{
     int x,y;
```

```
    printf("Enter first value i.e x : ");
    scanf("%d",&x);
    printf("Enter second value i.e y : ");
    scanf("%d",&y);
    swap(&x,&y);
    printf(" In the main program : \n");
    printf(" x = %d\n",x);
    printf(" y = %d\n",y);
}
swap(int *a, int *b)
{
    int t;
    printf(" In the swap function : \n");
    printf(" x  =  a  =  %d\n",*a);
    printf(" y  =  b  =  %d\n",*b);
    t=*a;
    *a=*b;
    *b=t;
    printf(" After swapping : \n");
    printf(" x  =  a  =  %d\n",*a);
    printf(" y  =  b  =  %d\n",*b);
}
```

OUTPUT :

```
Enter first value i.e. x : 33
Enter second value i.e. y : 64
In the swap function :
x  =  a  =  33
y  =  b  =  64
After interchanging :
x  =  a  =  64
y  =  b  =  33
In the main program :
x = 64
y = 33
```

Explanation : Here actual argument addresses in the calling function is copied into corresponding formal arguments i.e. address of x will be copied into a variable, address of y will be copied into b variable. The swap function will prints these a, b values before swapping and after swapping. Within the swap function a and b values are swapped and it will not affect the x and y also which are in the main function. Hence x and y values are also swapped.

9.6 PASSING ARRAY ELEMENTS TO A FUNCTION :

Array elements can be passed to a function by calling the function:

1. by value i.e. by passing values of array elements to the function.
2. by reference i.e. by passing addresses of array elements to the function.

Program 9.6 :

```
/*   PROGRAM TO ACCEPT A STATIC ARRAY AND PRINT IT BY
     CALL BY VALUE     */
#include<stdio.h>
main()

{
    int i;
    int a[5]={33,44,55,66,77};
    for(i=0;i<5;i++)
        write(i,a[i]);
}
write(int c, int n)
{
    printf("\ta[%d] = %d \n",c+1,n);
}
```

OUTPUT :

```
    a[1] = 33
    a[2] = 44
```

a[3] = 55
a[4] = 66
a[5] = 77

Explanation : Here the values of i and arrary elements will be send to write function.

Program 9.7 :

```
/* PROGRAM TO ACCEPT A STATIC ARRAY AND PRINT IT BY CALL BY REFERENCE */
#include<stdio.h>
main()
{
    int i;
    int a[5]={33,44,55,66,77};
    for(i=0;i<5;i++)
        write(i,&a[i]);
}
write(int c,int *n)
{
    printf("\ta[%d] = %d \n",c+1,*n);
}
```

OUTPUT :

a[1] = 33
a[2] = 44
a[3] = 55
a[4] = 66
a[5] = 77

Explanation : Here the values of i and address of arrary elements will be send to write function.

9.7 POINTERS AND ARRAYS :

Array elements are always stored in contiguous memory locations. A pointer incremented always points to an immediately next location of its type.

Program 9.8 :

```
// PROGRAM TO PRINT ARRAY ELEMENTS
#include<stdio.h>
main()
{
    int a[]={32,43,54,65,78},i,*j;
    j=&a[0];
    clrscr();
    for(i=0;i<5;i++)
    {
        printf("a[%d] = Value of address at %u = %d\n",i,j,*j);
        j++;
    }
}
```

OUTPUT :

a[0] = Value of address at 65516 = 32
a[1] = Value of address at 65518 = 43
a[2] = Value of address at 65520 = 54
a[3] = Value of address at 65522 = 65
a[4] = Value of address at 65524 = 78

Explanation : Here the values j is holding starting location address of array a i.e. address of array a. Whenever j is incremented by 1 it will holds the next location address of the array.

Passing an entire array to a function : We can pass the entire array to a function rather than individual elements.

Program 9.9 :

```
// PROGRAM TO PASS THE ENTIRE ARRAY TO FUNCTION
#include<stdio.h>
main()
{
```

```
        int a[]={32,43,54,65,78};
        display(&a[0],5);
}
display(int *i, int x)
{
        int j;
        for(j=0;j<x;j++)
        {
             printf("a[%d] = Value of address at %u = %d\n",j,i,*i);
             i++;
        }
}
```

OUTPUT :

a[0] = Value of address at 65516 = 32
a[1] = Value of address at 65518 = 43
a[2] = Value of address at 65520 = 54
a[3] = Value of address at 65522 = 65
a[4] = Value of address at 65524 = 78

Explanation : Here the display function is accepting the starting address of the array instead of accepting individual element addresses.

Accessing array element in different ways : We can access the array elements in different ways. Assume that the ith element of the array can be accessed :

num[i], *(num+i), *(i+num), i[num]

Ex:

Program 9.10 :

```
/*   ACCESSING ARRAY ELEMENT IN DIFFERENT WAYS    */
#include<stdio.h>
main()
{
        int num[]={12,23,34,45,56};
        int i;
        for(i=0;i<5;i++)
        {
             printf("a[%d] = Value of address at %u = ",i,&num[i]);
             printf("%d %d %d %d\n",num[i], *(num+i),*(i+num), i[num]);
        }
}
```

OUTPUT :

Address = 65490 Element = 12 12 12 12
Address = 65492 Element = 23 23 23 23
Address = 65494 Element = 34 34 34 34
Address = 65496 Element = 45 45 45 45
Address = 65498 Element = 56 56 56 56

Explanation : The expression num[i], i[num], *(num+i) and *(i+num) are same.

9.8 CHARACTER POINTERS AND FUNCTIONS :

If we wish to store "Computer". we can store it either in a string or some location in memory and assign the address of the string in a ***char*** pointer. This is as shown below:

```
char str[] = "Computer";
char *p = "Computer";
```

The difference between these two forms is : we can't assign a string to another, whereas, we can assign a ***char*** pointer to another char pointer. This is shown in the following program.

Program 9.11 :

```
/* PROGRAM TO ASSIGN ONE STRING TO ANOTHER AND ASSIGN A CHARACTER
POINTER TO ANOTHER   */
main()
{
        char str1[]="Computer" ;
        char str2[10];
        char *s = "Good Morning";
```

```
        char *q;
        str2 = str1;        /*  It is error   */
        q=s;                /* it will works   */
}
```

Explanation : During the compilation will show error i.e. one string can not assign to another.
Once a string has been defined it can't be initialized to another set of characters. Unlike strings, such operation is perfectly valid in ***char*** pointers
Program 9.12 :

```
/* PROGRAM TO DEFINE A STRING AND TRYING TO INITIALIZE THAT STRING
*/
main()
{
        char str1[] = " hai! ";
        char *p = " hai! ";
        str1 = " Bye ";          /* It is an error  */
        p = " Bye ";             /*   It will works   */
}
```

Explanation : During the compilation will show error i.e. once string has been defined it cannot initialize.

Program 9.13 :

```
/*   PROGRAM TO PRINT THE GIVEN STRING    */
#include<stdio.h>
main()
{
        char city[100];
        int i,l;
        printf("  Enter any city name  : ");
        scanf("%s",city);
        printf("  The given string is : ");
        prn(city);
}
prn(char *city)
{
        while(*city!='\0')
        {
                printf("%c",*city);
                city++;
        }
}
```

OUTPUT :

```
Enter any city name : Hyderabad
The given string is : Hyderabad
```

Explanation : The city name wil be passed to prn function by pointer i.e. city in prn function is a char pointer to hold the starting address of the string.

Program 9.14 :

```
/*   PROGRAM TO CALCULATE THE LENGTH OF THE GIVEN STRING    */
#include<stdio.h>
main()
{
        char city[100];
        int i,l;
        printf("  Enter any city name  : ");
        scanf("%s",city);
        l=len(city);
        printf("  The length of the given string is : %d\n",l);
}
len(char *city)
{
        int l1=0;
        while(*city!='\0')
```

```
    {
        ll++;
        city++;
    }
    return(ll);
}
```

OUTPUT :

Enter any city name : Bangalore
The length of the given string is : 9

Explanation : The city name wil be passed to len function by pointer i.e. city in len function is a char pointer to hold the starting address of the string.

9.9 ARRAY OF POINTERS :

Homogeneous collection of pointers is referred to array of pointers, i.e. coection of pointers of the same data type. The pointers are declared in array form like other data type. An integer pointer array marks of size can be declared as :

```
int *marks[20]
```

A pointer may also be used for more than one array declaration of same data type.
Ex :

```
int s1[20], s2[20], s3[20];
int *marks[10];
```

Now we can assign the addresses of first element to each array as :

```
marks[0] = &s1[0];
marks[1] = &s2[0];
marks[2] = &s3[0];
```

9.10 POINTER TO FUNCTION :

The function name itself represents the address of that function. Hence, it is possible to define a pointer to a function. Functions are stored in memory similar to data. A function can be assigned, placed in the arrays passed as an argument to other functions and returned by a function by means of a pointer to a function. The declaration of a pointer to a function is :

```
data_type(*function_name)(type1,type2, . . ., typen)
```

The function call has the format :

```
(*function_name)(arg1, arg2,...,argn)
```

The parentheses in (*function_name) are necessary, since in *function_name() the parentheses have higher precedence than * and they evaluated from left to right. If the parentheses are omitted, then it becomes a function returning a pointer. The general format for a function returning a pointer is :

```
*function_name(arg1, arg2, ..., argn)
```

Thus, the pointer to a function must be declared carefully to avoid the confusion between a function returning pointer and a pointer to a function. For example

```
double(*pf)();      /*  pf is a pointer to a function    */
double *fun();      /*  fun is function returning a pointer  */
```

Program 9.15 :

```
/*  PROGRAM THAT ILLUSTRATES POINTER TO A FUNCTION    */
main()
{
    int square(int),cubic(int);
    int(*arr[2])();          /*  pointer to function declaration  */
    arr[0]=square;
    arr[1]=cubic;
    printf("Square of %d is %d\n",5,(*ap[0](5));
    printf("Cubic value of %d is %d\n",5,(*ap[1])(5));
}
square(int x)
{
    return x*x;
}
cubic(int x)
{
    return x*x*x;
}
```

OUTPUT :

Sqaure fo 5 is 25

Cubic value of 5 is 125

9.11 POINTERS TO POINTERS :

Pointers to pointers is logically same as the array of pointers. Consider the following declaration :

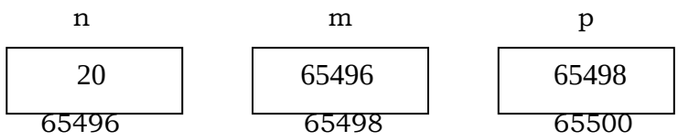
```
int n=20, *m;
m=&n;
```

Here m is not an ordinary variable like any other integer variable. It is a variable which contains the address of another variable n. The following memory map would illustrate the content of n and m.

Generally a pointer is a variable which contains address of another variable, this variable itself could be another pointer. Thus we have a pointer which contains another pointer's address. Consider the following declaration statements :

```
int n=20, *m, **p;
m=&n;
p=&m;
```

The above situation is called as pointer to pointer. First the address of n is assigned to m and the address of m is assigned to p. The expression *m gives the value of n and *p gives the value of m, which is the address of the variable n. Hence, the value of n can also be obtained by the expression **p.



Program 9.16 :

```
/*  PROGRAM TO DEMONSTRATE THE CONCEPT OF POINTERS TO POINTERS      */
#include<stdio.h>
main()
{
      int n=20;
      int *m;
      int **p;
      m=&n;
      p=&m;
      clrscr();
      printf("  Address of n is : %u\n",&n);
      printf("  Address of n is : %u\n",m);
      printf("  Address of n is : %u\n",*p);
      printf("  Address of m is : %u\n",&m);
      printf("  Address of m is : %u\n",p);
      printf("  Address of p is : %u\n",&p);
      printf("  Value of m is : %u\n",m);
      printf("  Value of p is : %u\n",p);
      printf("  Value of n is : %d\n",n);
      printf("  Value of n is : %d\n",*(&n));
      printf("  Value of n is : %d\n",*m);
      printf("  Value of n is : %d\n",**p);
}
```

OUTPUT :

```
Address of n is : 65496
Address of n is : 65496
Address of n is : 65496
Address of m is : 65498
Address of m is : 65498
```

Address of p is : 65500
Value of m is : 65496
Value of p is : 65498
Value of n is : 20
Value of n is : 20
Value of n is : 20
Value of n is : 20

Explanation : Here the last printf statement prints the value of n by using **p. Here p value is 65498 value_at_address(65498) is 65496, now value_at_address(65496) is n.

9.12 POINTERS AND MULTIDIMENSIONAL ARRAYS :

A multidimensional array can be accessed in different ways. as in a one dimensional array. The array name of a multidimensional array also denotes the address of the 0th element in that array. For example if the declaration int a[10][15], x[5][10][8]; the array names are a and x denote the addresses of the array variable a[0][0] and x[0][0][0] respectively. The following table illustrate the various expressions employed to refer to the elements in these arrays.

Address of a[0][0]	Value of a[0][0]	Address of a[3][4]	Value of a[3][4]	Address of a[i][j]	Value of a[i][j]
&a[0][0]	*&a[0][0]	&a[3][4]	*&a[3][4]	&a[i][j]	*&a[i][j]
a[0]	*a[0]	a[3]+4	*(a[3]+4)	a[i]+j	*(a[i]+j)
*a	**a	*(a+3)+4	*(*(a+3)+4)	*(a+i)+j	*(*(a+I)+j)
a	a[0][0]				*(a+i)+[j]

9.13 DYNAMIC MEMORY ALLOCATION :

During the compilation time the memory allocation performed is called static memory allocation. This static allocation has a disadvantage that, there is no way the size of data structure can be increased or decreased. If we declared the integer array m with the size of 100, the moment of declaration 200 bytes are reserved for storing 100 integers in it. If we are using 25 locations remaining memory will be wasted. If we want to store 101 elements there is no way.

The solution is to avoid static memory allocation is allocation of memory should be done at the time of running, it is called the dynamic memory allocation. The functions ***malloc*** and ***calloc*** is used for dynamic memory allocation.

The ***malloc*** function is used to allocate memory at run time. It is called with one parameter which specifies the number of bytes of memory required. Typecasting is necessary for malloc(), by default it returns a pointer to a void.

Ex :

```
int *ip;
ip = (int*) malloc(10*sizeof(int));
```

The above statements allocate space for 10 integers.

The ***calloc*** function is used to allocate memory at run time. It is called with two parameters, the first one is an unsigned element specifying the number of objects for which you want to allocate space, the second parameter specifies the size of each element.

The main difference between malloc and calloc is that, the calloc initializes the allocated space to zeros, malloc does not provide any initialization.

The main difference between malloc() and calloc() is that by default the memory is allocated by malloc() contains garbage values, where as that which allocated by calloc() contains all zeros. To use these function, include the file 'alloc.h' at the beginning of the program.

Program 9.17 :

```
/*PROGRAM TO EXPLAIN THE DYNAMIC MEMORY ALLOCATION */
#include<stdio.h>
main()
{
      int n, i, sum=0, avg, *marks;
      clrscr();
       printf("Enter how many students are there : ");
       scanf("%d",&n);
       marks=(int*)malloc(n*2);
```

```
 if(marks==NULL)
{
        printf(" Memory allocation unsuccessful \n");
        exit();
}
for(i=0;i<n;i++)
{
        printf(" Enter marks[%d] = ",i);
        scanf("%d",(marks+i));
        sum+=*(marks+i);
}
printf(" The students marks are : \n");
for(i=0;i<n;i++)
        printf("marks[%d] = %d\n",i,*(marks+i));
avg=sum/n;
printf("  Sum of all students marks is : %d\n",sum);
printf("  Average marks is : %d\n",avg);
}
```

Explanation : Here the memory allocation can be done at run time only by using malloc function.

9.14 COMMAND LINE ARGUMENTS :

The function man() in C can also accept arguments or parameters like the other functions. It has two arguments ***argc*** (for argument count) and ***argv*** (for argument vector). It may have one of the following two forms…

```
main(int argc, char *argv[])
            (or)
main(int argc, char **argv)
```

Here argc represents the number of arguments and argv is a pointer to an array of strings or pointer to pointer to character

The argument argv is used to pass strings to the programs. Hence, the arguments argc and argv are called as program parameters. After successful compilation, the program is executed by using an execution command. The execution command depends on the system running the program. It is possible to pass the values of program parameter argv through an execution command only. Since the values are available and obtained from the command line, they are also known as ***command line arguments.***

9.15 OBJECTIVE TYPE QUESTIONS

1. Pointers are supported in [ ]
   a) FORTRAN b) PASCAL
   c) C d) both B and C
2. Pointer variable may be assigned [ ]
   a) an address value represented in hexadecimal
   b) an address value represented in octal
   c) the address of another variable d) none
3. A pointer value refers to [ ]
   a) an integer constant b) a float value
   c) any valid address in memory d) any ordinary variable
4. Identify the correct declaration of variables p1,p2 [ ]
   a. int p1,p2 b. int *p1,p2
   c. int p1,*p2 d.int *p1, *p2
5. The operators exclusively used in connection with pointers are
   a) * and / b) & and * [ ]
   c) & and | d) - and >
6. Identify the invalid expression for given resiter int r=10; [ ]
   a) r=20 b) &r
   c) r+15 d) r/10
7. Identify the invalid expression [ ]
   a) &275 b) b.&a+b
   c) &(a*b) d) All the above
8. The operand of the address of pointer is [ ]
   a. a constant b.an expression
   c. a named region of storage d.a register variable
9. The address of operator returns [ ]
   a) The address of its operand b) Lvalue

c) Both a&b d) R value

10. After the execution of the statement in x; the value of x is [ ]
a) 0 b) undefined c) 1 d) -1

11. Identify the invalid pointer operator [ ]
a) & b) * c) >> d) none of the above

12. The number if arguments used in malloc() is [ ]
a) 0 b) 1 c) 2 d) 3

13. The number if arguments used in calloc() is [ ]
a) 0 b) 1 c) 2 d) 3

14. The number if arguments used in realloc() is [ ]
a) 0 b) 1 c) 2 d) 3

15. The function used for dynamic deallocation of memory is [ ]
a) destroy() b) delete() c) free() d) remove()

16. The pointers can be used to achieve [ ]
a) call by function b) call by reference
c) call by name d) call by procedure

ANSWERS

1.	d	2.	c	3.	c	4.	d	5.	b
6.	b	7.	d	8.	c	9.	a	10	b
11	c	12	b	13	c	14	c	15	c
16	b								

10 – STRUCTURES & UNIONS

10.1 INTRODUCTION :

A **structure** in C is a hetereogeneous data type, similar to the records of dBASE and Pascal. A structure can have its member of different types. A structure is a collection of variables referenced under one name. The arrays and structures are different from each other as (i) arrays are homogeneous in nature i.e., same type of elements, where as structures are heterogeneous i.e., different type of elements. (ii) array element is referred by its position, where as structure elements has a unique name.

10.2 STRUCTURE DECLARATION :

Structures help to organize complex data in a more meaningful way. Like other variables, a structure variable also have to be declared. The structre declaration starts with the keyword **struct** followed by a tag. The general format of a simple structure declaration is :

```
struct structure_name
{
        type var1;
        type var2;
          .
          .
        type varn;
};
```

10.3 STRUCTURE DEFINITION :

It is collection of logically related data items grouped together under a single name called structure tag. The data items that make up a structure are called its members, components, or fields and can be of different type.

Ex : Declaring a structure of name books with members of title, author, number of pages and price.

```
struct books
{
    char title[25];
    char author[20];
    int pages;
    float price;
};

struct books book1, book2, book3;
```

The above declaration may be written as :

```
struct books
{
    char title[25];
    char author[20];
    int pages;
    float price;
}   book1,book2,book3;
```

10.4 INITIALIZATION OF STRUCTURES :

Like other data types a structure variable can be initialized. A structure to be initialized must be either static or external.

```
main()
{
    static struct
    {
        char name[30];
        int age;
        float height;
    }
    student={"Srinivas",20,180.75};
    . . . .
    . . . .
 }
```

This assigns the string "Srinivas" to student.name, value 20 to student.age and 180.75 to student.height.

Suppose you want to initialize more than one structure variable:

```
main()
{
    struct sturec
    {
        char name[30];
        int age;
        float height;
```

```
        };
        static struct sturec student1={"Srinivas",20,180.75};
        static struct sturec student2={"Ramesh",22,177.25};
          . . . .
          . . . .
}
```

Another method is to initialize a structure variable outside the function

```
 struct sturec
 {
        char name[30];
        int age;
        float height;
 } student1={"Srinivas",20,189.76}
 main()
 {
        static sturec student2={"Ramesh",22,177.25};
          . . . .
          . . . .
 }
```

10.5 ACCESSING STRUCTURE ELEMENTS :

C provides the .(dot)/period operator using which structure members can be accessed independently. The dot operator connects a structure variable with its member. The dot operator must have a structure variable on its left and a legal member name on its right.

Ex :

```
student1.age=20;
```

Note that the structure tag is not a variable name. It is rather a name given to a templet of a structure. Consider the following statement is a not valid one.

```
sturec.age=20;
```

Program 10.1 :

```
/* DEFINE A STRUCTURE TYPE, STUREC, THAT CONTAIN NAME, AGE, HEIGHT. USING THIS STRUCTURE, WRITE A PROGRAM TO READ FOR ONE STUDENT DETAILS FROM THE KEYBOARD AND PRINT THE SAME ON THE SCREEN */
#include<stdio.h>
struct sturec
{
        char name[30];
        int age;
        float height;
};
 main()
{
    struct sturec student
    clrscr();
    printf("      Enter a student details : \n");
    printf("  Enter a student name :   ");
    scanf("%s",student.name);
    printf("  Enter a student age :  ");
    scanf("%d",&student.age);
    printf("  Enter a student height :  ");
    scanf("%f",student.height);
    printf("\n\n  Student's name is :  %s\n",student.name);
    printf("  Student's age is :  %d\n",student.age);
    printf("  Student's height is :  %f\n",student.salary);
}
```

OUTPUT:

```
Enter a student details :
Enter a student name :   Rajesh
Enter a student age :  21
Enter a student height :  180.30
Student's name is :  Rajesh
Student's age is :  21
Student's height is :  180.302314
```

10.6 COMPARISION OF STRUCTURE VARIABLES :

Like ordinary ariables two variables of the same structure type can be compared.

Program 10.2 :

```
/* PROGRAM TO ILLUSTRATE THE COMPARISION OF STRUCTURE
                                        VARIABLES */
#include<stdio.h>
struct sturec
{
      char name[30];
      int number;
      int marks;
 };
 main()
 {
    int x;
    static struct sturec stu1={"Nikhila",111,83};
    static struct sturec stu2={"Vignesh",222,85};
    struct sturec stu3;
    stu3=stu2;
    if(stu2.name==stu3.name && stu2.number==stu3.number &&
                        stu2.marks==stu3.marks)
          printf("\n Student2 and Student3 are same\n");
    else
          printf("\n student2 and student3 are different\n");
 }
```

OUTPUT:

Student2 and Student3 are same

10.7 ARRAYS OF STRUCTURES :

You can declare array of structures, each element of the array represents a structure variable.

Ex. :

```
struct stumarks student[100];
```

defines an array called 'student' that consists of 100 elements, Each element is defined to be of the type struct stumarks. Consider the following declaration.

```
struct emprec
 {
    char name[25];
    float basic;
    float da;
    float hra;
    float pf;
    float total;
 };
main()
 {
     static struct emprec emp[3]=
                {"Nagesh",8000.00,4000.00,1600.00,600.00,0.0},
                {"Ramaiah",6000.00,3000.00,1250.00,600.00,0.0},
                {"Kiran",10000.00,5400.00,2200.00,700.00,0.0};
        . . . .
        . . . .
 }
```

Program 10.3 :

```
/*      WRITE A PROGRAM TO CALCULATE THE TOTAL SALARY OF EACH
EMPLOYEE AND INDIVIDUAL TOTALS    */

# include<stdio.h>
struct emprec
 {
```

```
    char name[25];
    float basic;
    float da;
    float hra;
    float pf;
    float total;
};
main()
{
     int i;
     float tbasic=0.0, thra=0.0, tda=0.0, tpf=0.0,gtotal=0.0;
    static struct emprec emp[3]=
                {"Nagesh",8000.00,4000.00,1600.00,600.00,0.0},
                {"Ramaiah",6000.00,3000.00,1250.00,600.00,0.0},
                {"Kiran",10000.00,5400.00,2200.00,700.00,0.0};
     for(i=0;i<3;i++)
     {
          emp[i].total= emp[i].basic+emp[i].da+ emp[i].hra-emp[i].pf;
          tbasic=tbasic+emp[i].basic;
          tda=tda+emp[i].da;
          thra=thra+emp[i].hra;
          tpf=tpf+emp[i].pf;
          gtotal=gtotal+emp[i].total;
     }
     for(i=0;i<3;i++)
          printf("    stu[%d]    :    %f\n",i+1,emp[i].total);
     printf("Total Basic :  %f\n",tbasic);
     printf("Total DA :  %f\n",tda);
     printf("Total HRA :  %f\n",thra);
     printf("Total PF :  %f\n",tpf);
     printf("\n    Grand Total   :    %d\n", gtotal);
}
```

10.8 ARRAYS WITHIN STRUCTURES :

It is possible to have an array inside a function, i.e., an array as structure member. We can use single or multi dimensional array of type int or float also.

```
struct marks
{
    int no;
    int sub[5];
    int total;
} stu[10];
```

10.9 NESTED STRUCTURES :

The concept of nested structures refers to use of structures within a structure. The definition of C says that any legal data type can be a member of a structure, any legal data type includes a structure too.

```
 struct emprec
 {
     char name[30];
     int age;
     struct
     {
          int day;
          char month[20];
          int year;
     }j_date;
     float sal;
 }
```

10.10 STRUCTURES AND FUNCTIONS :

A structure can also be used in functions, while using the same structure in different functions. It is essential to pass the structure as an argument, the structure variable or a pointer to the structure may be passed. When the structure

variable is passed the changes made in the members of the structure available only in the called function(Call by Value). But, if a pointer to a structure is used, the changes are available in the calling function also (effect of Call by Reference). Based on the requirement the structure may be passed simply as structure variable or pointer to a structure. It is important to declare the parameters appropriately in the function definition.

A function definition shown below that contains the declaration of the structure in the left side and the declaration of a pointer to structure on the right side. The actual and formal parameters can have the same names or different names.

C consists of functions. The function main() subdivided into a number of modules. Hence, it is a good programming style if a program is written using many functions. Observe the declaration of the parameter in the function definition.

One :

```
main()
{
      typedef struct
      {
            ……
            …….
      }NODE;
      NODE s,*p = &s;
      ……….
      create(s);
      ……….
}
create (NODE start)
{
      .. …….
}
```

Two :

```
main()

{
      typedef struct
      {
            ……
            ……
      }NODE;
      NODE s,*p=&s;
      ……….
      create(p);
      ……….
}
create (NODE *start)
{
      .. …….
}
```

Three :

```
main()
 {
      typedef struct
      {
            int id;
            char name[30];
      }EMPLOYEE;
      EMPLOYEE emp;
      ……..
      organize(emp.id);              /*  member passed   */
      ……..
 }
```

```
organize(int x)
{
    .....
}
```

10.11 STRUCTURES AND POINTERS :

We specified that pointer is a variable that holds the memory address of other variables of basic datatypes such as int, float, char, etc., A pointer can also be used to hold the address of structure variables. Pointers along with structures are used to make linked list, trees, graphs, etc.,

Ex. :

```
struct emprec
 {
    char name[25];
    float basic;
    float da;
    float hra;
    float pf;
 };

struct emprec *emp;
```

In the above declaration, ***emp** is a pointer variable which holds the address of the structure emprec. Like other structure variables we can access the structure variables.

Ex. :

```
(*emp).basic
        (or)
emp->basic
```

In the second declaration, pointer to structure is expressed using a – (iphen) sign followed by greater than (>) sign.

10.12 SELF-REFERENTIAL STRUCTURE :

It is possible to have a member of a structure as a pointer to itself (structure itself as a member is not possible). In a structure, if one or more members are pointers pointing to the same structure, then this structure is known as a self-referential structure.

Ex :

```
typedef struct vertex
 {
    int data;
    struct vertex *np;
 }NODE;
```

Recursive declaration of a structure as shown above is allowed in C. Here, ***np*** is a member, which is a ***pointer to struct vertex.*** But it is illegal to use the structure itself as its member. One can appreciate the use of a pointer yielding a self-referential structure. There is also an indirect way of creating a self-referential structure as given below.

```
struct first
 {
    ....
    struct second *s;          /*  s  points to second  */
 };
struct second
 {
    :
    struct first *f;           /*   f   points to first    */
 };
```

The structure ***first*** has a member s pointing to the structure ***second,*** which has a member f pointing to the structure ***first*** **.**

An indirect recursive declaration of a pointer to a structure also results in a self-referential structure.

10.13 UNIONS :

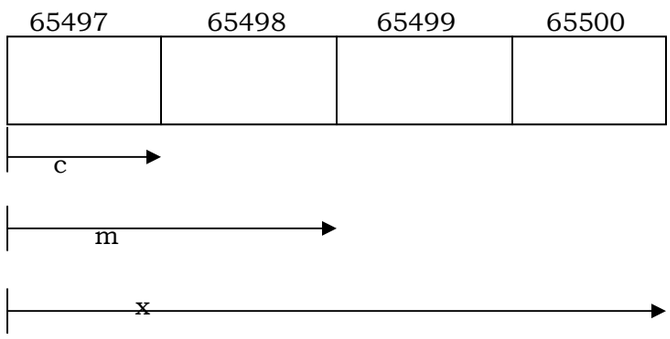
Unions are similar to structure data types. This union data type allows you to overlay more than one variable in the same memory area. Normally, every variable is stored in separate locations, thus each one has its own address. The major difference between the structure and union is in terms of storage. In structures each member has its own storage location, whereas all the members of a union use the same location. The general form of union declaration is :

```
union name
{
        type var1;
        type var2;
              .
              .
};
```

Ex:

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

In the above example a variable code declares the type union item. The union contains three members each with a different data type. However we can use only one of them at a time. This is due to the fact that only one location is allocated for a union variable, irrespective of its size.



The compiler allocates a piece of storage that is large enough to hold the largest variable type in the union. In the above declaration the member x requires 4 bytes which is the largest among the members. The above figure shows how all the three variables share the same address.

Accessing Union elements : To access a union member we can use the same syntax that we used in the structure members.

Ex: code.m
code.x

10.14 OBJECTIVE TYPE QUESTIONS

1. Structure is a [ ]
   a) scalar data type b) derived data type
   c) both a&b d) primitive data type
2. Structure is a data type in which [ ]
   a) each element must have the same element
   b) each element must have pointer type only
   c) each element must have different data type
   d) no element is defined
3. C provide a facility for user defined data type using [ ]
   a) pointer b) function c) structure d) array
4. The keyword used to represent a structure data type is [ ]
   a) strictire b) struct c) struc d) structr
5. Structure declaration [ ]
   a) describes the prototype b) creates structure variable
   c) define the structure function d) is not necessary
6. Structure definition [ ]
   a) describes the prototype b) creates structure variable
   c) define the structure function d) is not necessary

7. The operator used to access the structure member is [ ]
   a) * b) . c) [] d) &
8. The operator exclusively used with pointer to structure is [ ]
   a) . b) -> c) [] d) *
9. If one or more members of a structure are other structure is known as [ ]
   a) nested structure b) invalid structure
   c) self-referential structure d) unstructured
10. Union is [ ]
   a) a special type of strcture b) a pointer data type
   c) a function data type d) not a data type
11. What is not possible with union [ ]
   a) Array of union b) Pointer of union
   c) Self-referential union d) None of the above
12. Structure is used to implement the data structure [ ]
   a) Stack b) Queue c) Tree d) All the above
13. The node in a linked list are implemented using [ ]
   a) Self-referential structure b) Nested structure
   c) Array of structure d) Ordinary structure
14. A bit field is [ ]
   a) int b) float c) double d) all the above

ANSWERS

1.	b	2.	c	3.	c	4.	b	5.	a
6.	b	7.	b	8.	b	9.	a	10	a
11	d	12	d	13	a	14	a		

======

11 – FILE OPERATION

11.1 INTRODUCTION

Files are used to store large amount of data. Computer stores files on secondary storage devices such as harddisk, tapes, etc., A file is a group of related records. A employee file normally contains the records of each employee. A record is a group of related fields. Each field belongs to the same employee. These fields are composed of characters(or bytes). A field is a group of characters that conveys meaning. A character may be a digit, letter and special symbols. Computers manipulate and process these character as pattern of bits. Since computer can process only 0s and 1s. Byes are most commonly composed of eight bits. A bit (either 0 or 1) is the smallest data item in a computer.

11.2 TEXT FIES AND BINARY FILES :

Binary files are collection of 1s and 0s, where as ASCII or plain text files allocates a number to each letter, digit and symbol. A plain text files doesnot contains formatting codes, i.e. no fonts, bold, italic, headers and etc., ASCII is being replaced in many applications by Unicode, which uses 16 bits per character.

11.3 STREAMS :

A stream is an abstraction that either produces or consumes information. A stream is linked to a physical device by the I/O system. All streams behave in the same manner i.e. input stream can abstract many different kinds of input : from a disk file, a keyboard, or a network socket. Likewise an output stream may refer to the console, a disk file or a network connection.

Byte stream and character stream : Some programming languages like Java supports defines two types of streams : byte and character. Byte streams provides a convenient means for handling input and output of bytes. Byte streams are used when reading or writing binary data. Character streams provides a convenient for handling input and output of characters. In some cases character streams are more efficient than byte streams.

11.4 STANDARD I/O :

Standard I/O library functions are available in a file named ***stdio.h*** (standard input output header file). The following line is included in a C program whenever standard I/O functions are used.

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

It causes the contents of the header file to be included within the program. If it is not included in a program, the user must define and declare the necessary I/O functions.

I/O Functions : C supports many I/O functions. Some of the built-in functions are used to read the input from keyboard and display the output on the screen. Here we are specifying two important I/O functions, they are character oriented I/O functions and Formatted I/O functions.

Character oriented I/O functions: Character oriented I/O functions are used to read or write one character at a time. The standard C library provides several functions for this purpose. The simplest functions getchar( ) and putchar( ).

getchar( ): The functions ***getchar( )*** is used to read a single character from the keyboard. This function returns an integer value of the character in the machine's character code. If the system uses the ASCII character code, then the ASCII value of the character is returned. The general format for using this function is

```
variable = getchar( );
```

Here the variable may be either int or char data type. The ***getchar( )*** function does not take any argument.

putchar( ) :

The function ***putchar( )*** writes its character argument on the screen. The general format for using this function is:

```
putchar(ASCII_value);
```

Here the ASCII value may be represented by an integer constant or a variable or a character constant.

Formatted I/O Functions : Character oriented I/O functions use the values of the machine's character code for any character. This low level (e.g. ASCII code of a character) means of data transmission leads to other conversion functions whenever a numeric data is to be read or written. In such situations, the formatted I/O functions are very simple to use. The users may be interested in giving the inputs and getting the outputs in a particular format. For this purpose the formatted I/O functions are used.

scanf( ) : This functions is used to read input values from the keyboard. The general format is

scanf("format_string",address_list);

Here the format_string contains the required formatting specifications enclosed within double quotes. address_list contains the addresses of the memory locations where the input data is stored. The addresses are separated by commas.

The format_string includes conversion specifications directing the conversion of the next input data. The converted input is placed in the addresses of the variables given in the address_list. Conversion specification is written as % followed by a ***conversion character.*** The conversion character specifies the interpretation of the input data.

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

printf( ) : This function is used to write the output values on the screen. It will be used in two different ways

1) printf("Any string") : It is used to display the string enclosed within double quotes.
2) printf("format_string",list_variables) : Here format_string is the same as defined in the function scanf( ) or a combination of string and convertion sections. list_variables contains constants or variables or expressions or functions separated by commas.

11.5 FILE OPERATIONS :

The major basic operations on files that supports C is : naming a file, opening a file, reading data from a file, writing data to a file and closing a file. There are two distinct ways to perform a file operations in C. The first method is known as the *low level I/O* and uses UNIX system calls. The second method is referred to as the *high level I/O* operation and uses functions in C's standard I/O library.

The FILE pointer : A FILE is a data type (defined in stdio.h), which is used to operate on files. A FILE pointer is a pointer to a FILE. You may open several files simultaneously, when you want to perform I/O on a file you need to identify and specify which file you want to operate on. This is the purpose the FILE pointer variable serves.

Opening a File : To open a file, the stdio library provides a function **fopen**. The prototype of the function fopen is :

FILE* fopen(char* filename, char* mode)

This function expects two parameters. The first one is a character pointer to be pointing to a string contains the name of the file to be opened. The seond parameter is a character pointer to be pointing to a string contains the mode of operation. Mode an be one of te following :

Mode	Access
"r"	Read only; File must be present and readable; Soon after opening urrent position is set to top of file.
"w"	Overwrite. If file present contents are discarded, If file is absent it is created.
"a"	Append. If file present any output done gets appended. If file is absent it gets created. As soon as file is opened current position is set to the end of file.
"r+"	Update. File must present. Read as well as write access available. As soon as file is opened current position is set to top of file.
"w+"	Overwrite. It is same as "w" mode except both for reading and writing
"a+"	Append. It is same as "a" mode except both for reading and writing

This fopen function tries to open the said file in the said mode. If successful, returns a FILE pointer. If not successful, returns a NULL pointer.

The general format for declaring and opening a file:

FILE *fp;

fp=fopen("filename","mode");

The first statement declares the variable fp as a pointer to the data type FILE. The second statement opens the file, named filename and assigns an identifier to the FILE type pointer fp. This pointer which contains all the information about

the file is subsequently used as a communication link between the system and the program.

Closing a file: A file must be closed as soon as all operations on it have been completed. We have to close a file is when we want to reopen the same file in a different mode. fclose function is used to close a file.

```
fclose(file_pointer);
```

Ex:-
```
FILE *x1,*x2;
x1=fopen("salary",r);
x2=fopen("employee",w);
  . . . .
  . . . .
  . . . .
fclose(x1);
fclose(x2);
```

All files are closed automatically whenever a program terminates. However closing a file as soon as you are done with it is a good programming habit.

Character I/O on files : C provides a number of functions to perform character input/output operations one files.

getc functions : The **getc()** function reads characters from a file opened in read mode by **fopen()**. Function getc is functionally similar to getchar. It reads a character from the specified file and return an int containing the value read. On end of file getc returns EOF. Prototype of getc function is :

```
int getc(File *fp);
```

Ex :
```
ch=getc(fptr);
```

putc functions : The **putc()** function writes characters to a file opened in write mode by **fopen()**. Function putc is functionally similar to putchar. It has two parameters, a character expression and file pointer. Prototype of putc function is :

```
putc(int, File *fp);
```

Ex :
```
putc(ch,fptr);
```

Program 11.1 :

```
/*  WRITE A PROGRAM TO READ DATA FROM THE KEYBOARD,  WRITE IT  TO A
FILE CALLED IN, AGAIN READ THE SAME  DATA  FROM THE IN FILE AND
DISPLAY IT ON THE SCREEN   */
#include<stdio.h>
main()
{
   FILE *f1;
   char c;
   clrscr();
   printf(" Enter data into a file : \n");
   f1=fopen("IN","w");
   while((c=getchar())!=EOF)
       putc(c,f1);
   fclose(f1);
   printf("\nData from the file : \n");
   f1=fopen("IN","r");
   while((c=getc(f1))!=EOF)
        printf(" %c ",c);
   fclose(f1);
}
```

OUTPUT :

```
Enter data into a file :
I am a Software Engineer
Data from the file :
I am a Software Engineer
```

11.6 getw AND putw FUNCTIONS :

The getw and putw are integer oriented functions. They are similar to the getc and putc functions, and are used to read and write integer values. These functions would be useful when we deal with only integer data. The general form of

```
putw(integer, fp);
getw(fp);
```

Ex:-

Program 11.2 :

/* A file name DATA contains a series of integer numbers. Code a program to read these numbers and then write all odd numbers to a file to be called ODD and all even numbers to a file to be called EVEN. */

```
#include<stdio.h>
main()
 {
   FILE *f1,*f2,*f3;
   int num,i;
   clrscr();
   printf(" Contents of DATA file :\n");
   f1=fopen("DATA","w");
   for(i=1;i<=30;i++)
   {
       printf("Enter an integer number :  ");
       scanf("%d",&num);
       if(num==-1)
          break;
       putw(num,f1);
   }
   fclose(f1);
   f1=fopen("DATA","r");
   f2=fopen("ODD","w");
   f3=fopen("EVEN","w");
   while((num=getw(f1))!=EOF)
   {
        if(num%2==0)
            putw(num,f3);
        else
            putw(num,f2);
   }
   fclose(f1);
   fclose(f2);
   fclose(f3);
   f2=fopen("ODD","r");
   f3=fopen("EVEN","r");
   printf("\n\nContents of ODD file :\n\n");
   while((num=getw(f2))!=EOF)
       printf(" %4d",num);
   printf("\n\nContents of EVEN file :\n\n");
   while((num=getw(f3))!=EOF)
       printf(" %4d",num);
   fclose(f2);
   fclose(f3);
 }
```

OUTPUT :

```
Enter an integer number :
Enter an integer number :  5
Enter an integer number :  10
Enter an integer number :  15
Enter an integer number :  20
```

```
Enter an integer number :  25
Enter an integer number :  30
Enter an integer number :  35
Enter an integer number :  40
Enter an integer number :  45
Enter an integer number :  50
Enter an integer number :  55
Enter an integer number :  60
Contents of ODD file :
      5        15      25      35      45      55
Contents of EVEN file :
      10       20      30      40      50      60
```

11.7 THE fprintf AND fscanf FUNCTIONS :

fscanf fuction : This function can be used to different data types from the specified file. It works in the similar way of scanf function. The general form is :

```
fscanf(fp,"control string",list);
```

Here the statement reads the variable list from the file referred by the file pointer by using the forats given in control string.

Ex:-

```
fscanf(fp,"%d%f",&age,&sal);
```

fprintf function : This function can be used to print different types of data. It works in the similar way of printf function. The general form is :

```
fprintf(fp,"control string",list)
```

Program 11.3 :

```
/*  Write a program to open a file named INVENTORY and store
    it the following data
    Item name     number        price          quantity
     TV             111          25000.75        15
     VCP            113          42000.00         3
     VCR            123          50000.35        10
Extend the program to read this data from the file INVENTORY      and display the
inventory table with the value of each item.
*/
#include<stdio.h>
main()
 {
    FILE *fp;
    int num,qty,i;
    float price, value;
    char item[10],filename[20];
    printf(" Enter the name : ");
    scanf("%s",filename);
    fp=fopen(filename,"w");
    printf(" Input inventory data : ");
    printf(" Item name  number  price   quantity ");
    for(i=1;i<=3;i++)
        fscanf(stdin,"%s%d%f%d",item,&num,&price,&qty);
    fclose(fp);
    fprintf(stdout,"\n");
    fp=fopen(filename,"r");
    printf(" Item name number price  quantity value");
    for(i=1;i<=3;i++)
    {
        fscanf(fp,"%s%d%f%d",item,&num,&price,&qty);
        value=price*qty;
        fprintf(stdout,"%s %d %f %d  %f\n",item,num,price,qty,value);
    }
    fclose(fp);
 }
```

11.8 ERROR HANDLING :

When the program encounter an error, the C library functions set a variable called **errno** (error number). The variable **errno** is an integer declared in the file **errno.h**. The C library functions deposit different values in the variable **errno** for different kinds of errors. When a program encounters an error it can inspect the value available in the variable **errno** and based on this decide on a suitable course of action. The program will have to inclue the file **errno.h** to use these symbolic constants. Some values deposited in **errno** for various error conditions are shown below :

errno vaue	Meaning
EACCES	Access denied
ENOENT	No such file or directory
ENOMEM	No room in RAM
ENOSPC	Disk full
EMFILE	Too many open files

When an error occurs, the program try to transmit error essages to the standard error device rather than to the standard output. This is to ensure that te output message, which is meant for the user's inspection, does not get redirected to a file or drain down a pipeline. The **stdio.h** header file, introduces three FILE pointer constants namely, **stdin**, **stdout** and **stderr**. These FILE pointer constants point to the standard input, standard output and standard error devices, respectively.

```
getc(stdin)                /* equivalent to getchar()   */
fputs("Computer\n",stdout)  /* same as puts("Computer")
```

The **stderr** FILE pointer is particularly useful with respect to error handling.

11.9 OBJECTIVE TYPE QUESTIONS

1. A Stream is [ ]
   a) a library function b) a system call
   c) a source or destination of data that may be associated with a disk or other I/O devices d) a file
2. I/O Stream can be [ ]
   a) a text stream b) a binary steam
   c) both a & b d) an I/O operation
3. FILE defined in stdio.h is [ ]
   a) a region of storage b) a data type
   c) not a data type d) a variable
4. A file pointer is [ ]
   a) a stream pointer b) a buffer pointer
   c) a pointer to FILE data type d) all the above

ANSWERS:

1. c 2. c 3. b 4. d

======