GE6151-COMPUTER PROGRAMMING

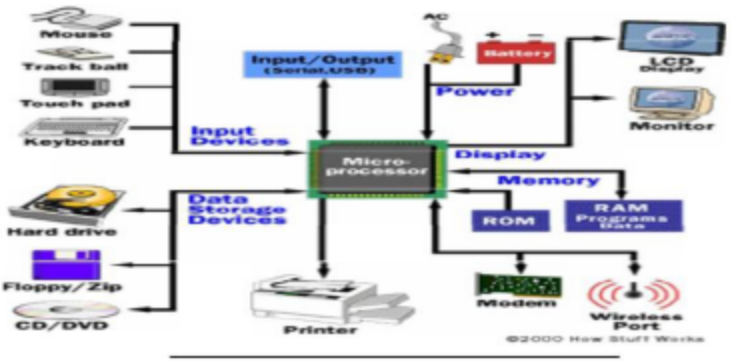
UNIT - 1

INTRODUCTION

Generation and Classification of Computers- Basic Organization of a Computer –Number System –Binary – Decimal – Conversion – Problems. Need for logical analysis and thinking – Algorithm – Pseudo code – Flow Chart.

INTRODUCTION

PARTS OF A COMPUTER



GENERATIONS OF COMPUTERS

The Zeroth Generation

The term Zeroth generation is used to refer to the period of development of computing, which predated the commercial production and sale of computer equipment. The period might be dated as extending from the mid-1800s. In particular, this period witnessed the emergence of the first electronics digital computers on the ABC, since it was the first to fully implement the idea of the stored program and serial execution of instructions. The development of EDVAC set the stage for the evolution of commercial computing and operating system software. The hardware component technology of this period was electronic vacuum tubes. The actual operation of these early computers took place without be benefit of an operating system. Early programs were written in machine language and each contained code for initiating operation of the computer itself. This system was clearly inefficient and depended on the varying competencies of the individual programmer as operators.

The First Generation, 1951-1956

The first generation marked the beginning of commercial computing. The first generation was characterized by high-speed vacuum tube as the active component technology. Operation continued without the benefit of an operating system for a time. The mode was called "closed shop" and was characterized by the appearance of hired operators who would select the job to be run, initial program load the system, run the user's program, and then select another job, and so forth. Programs began to be written in higher level, procedure-oriented languages, and thus the operator's routine expanded. The operator now selected a job, ran the translation program to assemble or compile the source program, and combined the translated object program along with any existing library programs that the program might need for input to the linking program, loaded and ran the composite linked program, and then handled the next job in a similar fashion. Application programs were run one at a time, and were translated with absolute computer addresses. There was no provision for moving a program to different location in storage for any reason. Similarly, a program bound to specific devices could not be run at all if any of these devices were busy or broken.

At the same time, the development of programming languages was moving away from the basic machine languages; first to assembly language, and later to procedure oriented languages, the most significant being the development of FORTRAN

The Second Generation, 1956-1964

The second generation of computer hardware was most notably characterized by transistors replacing vacuum tubes as the hardware component technology. In addition, some very important changes in hardware and software architectures occurred during this period. For the most part, computer systems remained card and tape-oriented systems. Significant use of random access devices, that is, disks, did not appear until towards the end of the second generation. Program processing was, for the most part, provided by large centralized computers operated under mono-programmed batch processing operating systems.

The most significant innovations addressed the problem of excessive central processor delay due to waiting for input/output operations. Recall that programs were executed by processing the machine instructions in a strictly sequential order. As a result, the CPU, with its high speed electronic component, was often forced to wait for completion of I/O operations which involved mechanical devices (card readers and tape drives) that were order of magnitude slower.These hardware developments led to enhancements of the operating system. I/O and data channel communication and control became functions of the operating system, both to relieve the application. programmer from the difficult details of I/O programming and to protect the integrity of the system to provide improved service to users by segmenting jobs and running shorter jobs first (during "prime time") and relegating longer jobs to lower priority or night time runs. System libraries became more widely available and more comprehensive as new utilities and application software components were available to programmers.

The second generation was a period of intense operating system development. Also it was the period for sequential batch processing. Researchers began to experiment with multiprogramming and multiprocessing.

The Third Generation, 1964-1979

The third generation officially began in April 1964 with IBM's announcement of its System/360 family of computers. Hardware technology began to use integrated circuits (ICs) which yielded significant advantages in both speed and economy. Operating System development continued with the introduction and widespread adoption of multiprogramming. This marked first by the appearance of more sophisticated I/O buffering in the form of spooling operating systems. These systems worked by introducing two new systems programs, a system reader to move input jobs from cards to disk, and a system writer to move job output from disk to printer, tape, or cards.

The spooling operating system in fact had multiprogramming since more than one program was resident in main storage at the same time. Later this basic idea of multiprogramming was extended to include more than one active user program in memory at time. To accommodate this extension, both the scheduler and the dispatcher were enhanced. In addition, memory management became more sophisticated in order to assure that the program code for each job or at least that part of the code being executed was resident in main storage. Users shared not only the system' hardware but also its software resources and file system disk space.

The third generation was an exciting time, indeed, for the development of both computer hardware and the accompanying operating system. During this period, the topic of operating systems became, in reality, a major element of the discipline of computing.

The Fourth Generation, 1979 - Present

The fourth generation is characterized by the appearance of the personal computer and the workstation. Miniaturization of electronic circuits and components continued and Large Scale Integration (LSI), the component technology of the third generation, was replaced by Very Large

Scale Integration (VLSI), which characterizes the fourth generation. However, improvements in hardware miniaturization and technology have evolved so fast that we now have inexpensive workstation-class computer capable of supporting multiprogramming and time-sharing. Hence the operating systems that supports today's personal computers and workstations look much like those which were available for the minicomputers of the third generation. Examples are Microsoft's DOS for IBM-compatible personal computers and UNIX for workstation. However, many of these desktop computers are now connected as networked or distributed systems. Computers in a networked system each have their operating system augmented with communication capabilities that enable users to remotely log into any system on the network and transfer information among machines that are connected to the network. The machines that make up distributed system operate as a virtual single processor system from the user's point of view; a central operating system controls and makes transparent the location in the system of the particular processor or processors and file systems that are handling any given program.

CLASSIFICATION OF COMPUTERS

There are four classifications of digital computer systems:

super-computer, mainframe computer, minicomputer, and microcomputer.

- Super-computers are very fast and powerful machines. Their internal architecture enables them to run at the speed of tens of MIPS (Million Instructions per Second). Super-computers are very expensive and for this reason are generally not used for CAD applications. Examples of super-computers are: Cray and CDC Cyber 205.

- Mainframe computers are built for general computing, directly serving the needs of business and engineering. Although these computing systems are a step below super-computers, they are still very fast and will process information at about 10 MIPS. Mainframe computing systems are located in a centralized computing center with 20-100+ workstations. This type of computer is still very expensive and is not readily found in architectural/interior design offices.

- Minicomputers were developed in the 1960's resulting from advances in microchip technology. Smaller and less expensive than mainframe computers, minicomputers run at several MIPS and can support 5-20 users. CAD usage throughout the 1960's used minicomputers due to their low cost and high performance. Examples of minicomputers are: DEC PDP, VAX 11.

- Microcomputers were invented in the 1970's and were generally used for home computing and dedicated data processing workstations. Advances in technology have improved microcomputer capabilities, resulting in the explosive growth of personal computers in industry. In the 1980's many medium and small design firms were finally introduced to CAD as a direct result of the low cost and availability of microcomputers. Examples are: IBM, Compaq, Dell, Gateway, and Apple Macintosh.

The average computer user today uses a microcomputer. These types of computers include PC's, laptops, notebooks, and hand-held computers such as Palm Pilots. Larger computers fall into a mini-or mainframe category. A mini-computer is 3-25 times faster than a micro. It is physically larger and has a greater storage capacity.

A mainframe is a larger type of computer and is typically 10-100 times faster than the micro. These computers require a controlled environment both for temperature and humidity. Both the mini and mainframe computers will support more workstations than will a micro. They also cost a great deal more than the micro running into several hundred thousand dollars for the mainframes.

Processors
The term processor is a sub-system of a data processing system which processes received information after it has been encoded into data by the input sub-system. These data are then processed by the processing sub-system before being sent to the output sub-system where they are decoded back into information. However, in common parlance processor is usually referred to the microprocessor, the brains of the modern day computers.

There are two main types of processors: CISC and RISC.

CISC: A Complex Instruction Set Computer (CISC) is a microprocessor Instruction Set Architecture (ISA) in which each instruction can indicate several low-level operations, such as a load from memory, an arithmetic operation, and a memory store, all in a single instruction. The term was coined in contrast to Reduced Instruction Set Computer (RISC).

Examples of CISC processors are the VAX, PDP-11, Motorola 68000 family and the Intel x86/Pentium CPUs.

RISC: Reduced Instruction Set Computing (RISC), is a microprocessor CPU design philosophy that favors a smaller and simpler set of instructions that all take about the same amount of time to execute. Most types of modern microprocessors are RISCs, for instance ARM, DEC Alpha, SPARC, MIPS, and PowerPC.

The microprocessor contains the CPU which is made up of three components--the control unit supervises all that is going on in the computer, the arithmetic/logic unit which performs the math and comparison operation, and temporary memory. Because of the progress in developing better microprocessors, computers are continually evolving into faster and better units.

Notebooks:
A laptop computer (also known as notebook computer) is a small mobile personal computer, usually weighing around from 1 to 3 kilograms (2 to 7 pounds). Notebooks smaller than an A4 sheet of paper and weighing around 1 kg are sometimes called sub-notebooks and those weighing around 5 kg a desk note (desktop/notebook). Computers larger than PDAs but smaller than notebooks are also sometimes called "palmtops". Laptops usually run on batteries.

Notebook Processor:
A notebook processor is a CPU optimized for notebook computers. All computing devices require a CPU. One of the main characteristics differentiating notebook processors from other CPUs is low-power consumption.
The notebook processor is becoming an increasing important market segment in the semiconductor industry. Notebook computers are an increasingly popular format of the broader category of mobile computers. The objective of a notebook computer is to provide the performance and functionality of a desktop computer in a portable size and weight. Wireless networking and low power consumption are primary consideration in the choice of a notebook processor.

Integrated Components
Unlike a desktop computer, a notebook has most of the components built-in or integrated into the computer. For desktop systems, determining which computer to buy is generally not based on what type of keyboard or mouse that is available. If you don't like the keyboard or mouse, you can always purchase something else. However, in the case of a notebook computer, the size of the keyboard or type of pointing device may be something that you need to consider unless you intend to use a regular mouse or full-sized keyboard. There are some notebooks that have a keyboard that expands when the notebook is opened which is a nice feature if you find the normal keyboard to be too small. Pointing devices vary from a touch pad to a stick within the keyboard to a roller or track-ball. Most notebooks have the video, sound, and speakers integrated into the computer and some notebooks even have a digital camera built-in which is very handy for video conferencing.

BOOTING:
In computing, booting is a bootstrapping process that starts operating systems when the user turns on a computer system. A boot sequence is the set of operations the computer performs when it is switched on which load an operating system.

Everything that happens between the times the computer switched on and it is ready to accept commands/input from the user is known as booting.

The process of reading disk blocks from the starting of the system disk (which contains the Operating System) and executing the code within the bootstrap. This will read further information off the disk to bring the whole operating system online. Device drivers are contained within the bootstrap code that support all the locally attached peripheral devices and if the computer is connected to a network, the operating system will transfer to the Network Operating system for the "client" to log onto a server

The Process of loading a computer memory with instructions needed for the computer to operate. The process and functions that a computer goes through when it first starts up, ending in the proper and complete loading of the Operating System. The sequence of computer operations from power-up until the system is ready for use

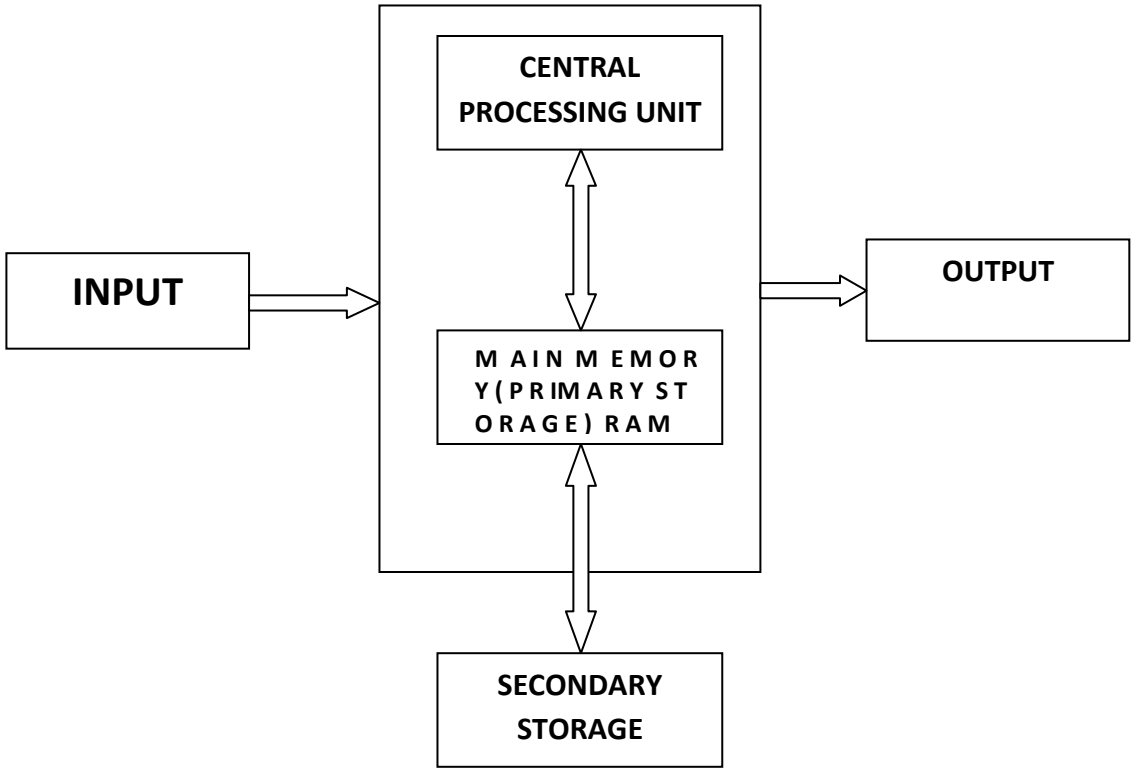
COLD BOOTING:
The cold booting is the situation, when all the computer peripherals are OFF and we start the computer by switching ON the power.
WARM BOOTING:
The warm booting is the situation, when we restart the computer by pressing the RESET button and pressing CTRL+ ALT + DEL keys together.

Graphic User Interface (GUI)
A program interface that takes advantage of the computer's graphics capabilities to make the program easier to use. Well-designed graphical user interfaces can free the user from learning complex command languages.On the other hand, many users find that they work more effectively with a command-driven interface, especially if they already know the command language.

BASIC COMPUTER ORGANIZATION:

A standard fully featured desktop configuration has basically four types of featured devices

1. Input Devices 2. Output Devices 3. Memory 4. Storage Devices



Introduction to CPU

- CPU
- The Arithmetic / Logic Unit (ALU)
- The Control Unit
- Main Memory
- External Memory
- Input / Output Devices
- The System Bus

CPU OPERATION

The fundamental operation of most CPUs
- To execute a sequence of stored instructions called a program.

1. The program is represented by a series of numbers that are kept in some kind of computer memory.
2. There are four steps that nearly all CPUs use in their operation: fetch, decode, execute, and write back.
3. Fetch:
o Retrieving an instruction from program memory.
o The location in program memory is determined by a program counter (PC)
o After an instruction is fetched, the PC is incremented by the length of the instruction word in terms of memory units.

Decode :

1.The instruction is broken up into parts that have significance to other portions of the CPU.

2.The way in which the numerical instruction value is interpreted is defined by the CPU's instruction set architecture (ISA).

3.Opcode, indicates which operation to perform.

4.The remaining parts of the number usually provide information required for that instruction, such as operands for an addition operation.

5.Such operands may be given as a constant value or as a place to locate a value: a register or a memory address, as determined by some addressing mode.

Execute :

1.During this step, various portions of the CPU are connected so they can perform the desired operation.
2.If, for instance, an addition operation was requested, an arithmetic logic unit (ALU) will be connected to a set of inputs and a set of outputs.
3.The inputs provide the numbers to be added, and the outputs will contain the final sum.

4. If the addition operation produces a result too large for the CPU to handle, an arithmetic overflow flag in a flags register may also be set.

Write back :

1.Simply "writes back" the results of the execute step to some form of memory.

2.Very often the results are written to some internal CPU register for quick access by subsequent instructions.

3.In other cases results may be written to slower, but cheaper and larger, main memory.

Some types of instructions manipulate the program counter rather than directly produce result data.

INPUT DEVICES

Anything that feeds the data into the computer. This data can be in alpha-numeric form which needs to be keyed-in or in its very basic natural form i.e. hear, smell, touch, see; taste & the sixth sense …feel?

Typical input devices are:

1. Keyboard
2. Mouse
3. Joystick
4. Digitizing Tablet
5. Touch Sensitive Screen
6. Light Pen
7. Space Mouse
8. Digital Stills Camera
9. Magnetic Ink Character Recognition (MICR)
10. OpticalMarkReader (OMR)
11. Image Scanner
12. Bar Codes
13. Magnetic Reader
14. Smart Cards
15. Voice Data Entry
16. Sound Capture
17. Video Capture

The **Keyboard** is the standard data input and operator control device for a computer. It consists of the standard QWERTY layout with a numeric keypad and additional function keys for control purposes.

The **Mouse** is a popular input device. You move it across the desk and its movement is shown on the screen by a marker known as a 'cursor'. You will need to click the buttons at the top of the mouse to select an option.

Track ball looks like a mouse, as the roller is on the top with selection buttons on the side. It is also a pointing device used to move the cursor and works like a mouse. For moving the cursor in a particular direction, the user spins the ball in that direction. It is sometimes considered better than a mouse, because it requires little arm movement and less desktop space. It is generally used with Portable computers.

Magnetic Ink Character Recognition (MICR) is used to recognize the magnetically charged characters, mainly found on bank cheques. The magnetically charged characters are written by special ink called magnetic ink. MICR device reads the patterns of these characters and compares them with special patterns stored in memory. Using MICR device, a large volume of cheques can be processed in a day. MICR is widely used by the banking industry for the processing of cheques.

The joystick is a rotary lever. Similar to an aircraft's control stick, it enables you to move within the screen's environment, and is widely used in the computer games industry.

A **Digitising Tablet** is a pointing device that facilitates the accurate input of drawings and designs. A drawing can be placed directly on the tablet, and the user traces outlines or inputs coordinate positions with a hand-held stylus.

A **Touch Sensitive Screen** is a pointing device that enables the user to interact with the computer by touching the screen. There are three types of Touch Screens: pressure-sensitive, capacitive surface and light beam.

A **Light Pen** is a pointing device shaped like a pen and is connected to a VDU. The tip of the light pen contains a light-sensitive element which, when placed against the screen, detects the light from the screen enabling the computer to identify the location of the pen on the screen. Light pens have the advantage of 'drawing' directly onto the screen, but this can become uncomfortable, and they are not as accurate as digitising tablets.

The **Space mouse** is different from a normal mouse as it has an X axis, a Y axis and a Z axis. It can be used for developing and moving around 3-D environments.

Digital Stills Cameras capture an image which is stored in memory within the camera. When the memory is full it can be erased and further images captured. The digital images can then be downloaded from the camera to a computer where they can be displayed, manipulated or printed.

The Optical Mark Reader (OMR) can read information in the form of numbers or letters and put it into the computer. The marks have to be precisely located as in multiple choice test papers.

Scanners allow information such as a photo or text to be input into a computer. Scanners are usually either A4 size (flatbed), or hand-held to scan a much smaller area. If text is to be scanned, you would use an Optical Character Recognition (OCR) program to recognise the printed text and then convert it to a digital text file that can be accessed using a computer.

A **Bar Code** is a pattern printed in lines of differing thickness. The system gives fast and error-free entry of information into the computer. You might have seen bar codes on goods in supermarkets, in libraries and on magazines. Bar codes provide a quick method of recording the sale of items.

Card Reader: This input device reads a magnetic strip on a card. Handy for security reasons, it provides quick identification of the card's owner. This method is used to run bank cash points or to provide quick identification of people entering buildings.

Smart Card: This input device stores data in a microprocessor embedded in the card. This allows information, which can be updated, to be stored on the card. This method is used in store cards which accumulate points for the purchaser, and to store phone numbers for cellular phones.

OUTPUT DEVICES :

Output devices display information in a way that you can you can understand. The most common output device is a monitor. It looks a lot a like a TV and houses the computer screen. The monitor allows you to 'see' what you and the computer are doing together.

Brief of Output Device

Output devices are pieces of equipment that are used to get information or any other response out from computer. These devices display information that has been held or generated within a computer. Output devices display information in a way that you can understand. The most common output device is a monitor.

Types of Output Device

Printing:		Plotter, Printer
Sound	:	Speakers
Visual	:	Monitor

A **Printer** is another common part of a computer system. It takes what you see on the computer screen and prints it on paper. There are two types of printers; Impact Printers and Non-Impact Printers.

Speakers are output devices that allow you to hear sound from your computer. Computer speakers are just like stereo speakers. There are usually two of them and they come in various sizes.

MEMORY OR PRIMARY STORAGE :

Purpose of Storage

The fundamental components of a general-purpose computer are arithmetic and logic unit, control circuitry, storage space, and input/output devices. If storage was removed, the device we had would be a simple calculator instead of a computer. The ability to store instructions that form a computer program, and the information that the instructions manipulate is what makes stored program architecture computers versatile.

Primary Storage

Primary storage is directly connected to the central processing unit of the computer. It must be present for the CPU to function correctly, just as in a biological analogy the lungs must be present (for oxygen storage) for the heart to function (to pump and oxygenate the blood). As shown in the diagram, primary storage typically consists of three kinds of storage:

Processors Register

It is the internal to the central processing unit. Registers contain information that the arithmetic and logic unit needs to carry out the current instruction. They are technically the fastest of all forms of computer storage.

Main memory

It contains the programs that are currently being run and the data the programs are operating on. The arithmetic and logic unit can very quickly transfer information between a processor register and locations in main storage, also known as a "memory addresses". In modern computers, electronic solid-state random access memory is used for main storage, and is directly connected to the CPU via a "memory bus" and a "data bus".

Cache memory

It is a special type of internal memory used by many central processing units to increase their performance or "throughput". Some of the information in the main memory is duplicated in the cache memory, which is slightly slower but of much greater capacity than the processor registers, and faster but much smaller than main memory.

Memory

Memory is often used as a shorter synonym for Random Access Memory (RAM). This kind of memory is located on one or more microchips that are physically close to the microprocessor in your computer. Most desktop and notebook computers sold today include at least 512 megabytes of RAM (which is really the minimum to be able to install an operating system). They are upgradeable, so you can add more when your computer runs really slowly.

The more RAM you have, the less frequently the computer has to access instructions and data from the more slowly accessed hard disk form of storage. Memory should be distinguished from storage, or the physical medium that holds the much larger amounts of data that won't fit into RAM and may not be immediately needed there.

Storage devices include hard disks, floppy disks, CDROMs, and tape backup systems. The terms auxiliary storage, auxiliary memory, and secondary memory have also been used for this kind of data repository.

RAM is temporary memory and is erased when you turn off your computer, so remember to save your work to a permanent form of storage space like those mentioned above before exiting programs or turning off your computer.

TYPES OF RAM:

There are two types of RAM used in PCs - Dynamic and Static RAM.

Dynamic RAM (DRAM): The information stored in Dynamic RAM has to be refreshed after every few milliseconds otherwise it will get erased. DRAM has higher storage capacity and is cheaper than Static RAM.

Static RAM (SRAM): The information stored in Static RAM need not be refreshed, but it remains stable as long as power supply is provided. SRAM is costlier but has higher speed than DRAM.

Additional kinds of integrated and quickly accessible memory are Read Only Memory (ROM), Programmable ROM (PROM), and Erasable Programmable ROM (EPROM). These are used to keep special programs and data, such as the BIOS, that need to be in your computer all the time. ROM is "built-in" computer memory containing data that normally can only be read, not written to (hence the name read only).

ROM contains the programming that allows your computer to be "booted up" or regenerated each time you turn it on. Unlike a computer's random access memory (RAM), the data in ROM is not lost when the computer power is turned off. The ROM is sustained by a small long life battery in your computer called the CMOS battery. If you ever do the hardware setup procedure with your computer, you effectively will be writing to ROM. It is non volatile, but not suited to storage of large quantities of data because it is expensive to produce. Typically, ROM must also be completely erased before it can be rewritten,

PROM (Programmable Read Only Memory)

A variation of the ROM chip is programmable read only memory. PROM can be programmed to record information using a facility known as PROM-programmer. However once the chip has been programmed the recorded information cannot be changed, i.e. the PROM becomes a ROM and the information can only be read.

EPROM (Erasable Programmable Read Only Memory)

As the name suggests the Erasable Programmable Read Only Memory, information can be erased and the chip programmed a new to record different information using a special PROM-Programmer. When EPROM is in use information can only be read and the information remains on the chip until it is erased.

STORAGE DEVICES

The purpose of storage in a computer is to hold data or information and get that data to the CPU as quickly as possible when it is needed. Computers use disks for storage: hard disks that are located inside the computer, and floppy or compact disks that are used externally.

- Computers Method of storing data & information for long term basis i.e. even after PC is switched off.
- It is non - volatile
- Can be easily removed and moved & attached to some other device
- Memory capacity can be extended to a greater extent
- Cheaper than primary memory

Storage Involves Two Processes

a) Writing data b) Reading data

Floppy Disks

The floppy disk drive (FDD) was invented at IBM by Alan Shugart in 1967. The first floppy drives used an 8-inch disk (later called a "diskette" as it got smaller), which evolved into the 5.25-inch disk that was used on the first IBM Personal Computer in August 1981. The 5.25-inch disk held 360 kilobytes compared to the 1.44 megabyte capacity of today's 3.5-inch diskette.

The 5.25-inch disks were dubbed "floppy" because the diskette packaging was a very flexible plastic envelope, unlike the rigid case used to hold today's 3.5-inch diskettes.

By the mid-1980s, the improved designs of the read/write heads, along with improvements in the magnetic recording media, led to the less-flexible, 3.5-inch, 1.44-megabyte (MB) capacity FDD in use today. For a few years, computers had both FDD sizes (3.5-inch and 5.25-inch). But by the mid-1990s, the 5.25-inch version had fallen out of popularity, partly because the diskette's recording surface could easily become contaminated by fingerprints through the open access area.

When you look at a floppy disk, you'll see a plastic case that measures 3 1/2 by 5 inches. Inside that case is a very thin piece of plastic that is coated with microscopic iron particles. This disk is much like the tape inside a video or audio cassette. Basically, a floppy disk drive reads and writes data to a small, circular piece of metal-coated plastic similar to audio cassette tape.

At one end of it is a small metal cover with a rectangular hole in it. That cover can be moved aside to show the flexible disk inside. But never touch the inner disk - you could damage the data that is stored on it. On one side of the floppy disk is a place for a label. On the other side is a silver circle with two holes in it. When the disk is inserted into the disk drive, the drive hooks into those holes to spin the circle. This causes the disk inside to spin at about 300 rpm! At the same time, the silver metal cover on the end is pushed aside so that the head in the disk drive can read and write to the disk.

Floppy disks are the smallest type of storage, holding only 1.44MB.

3.5-inch Diskettes (Floppy Disks) features:

- Spin rate: app. 300 revolutions per minute (rpm)
- High density (HD) disks more common today than older, double density (DD) disks
- Storage Capacity of HD disks is 1.44 MB

Floppy Disk Drive Terminology

Floppy disk - Also called diskette. The common size is 3.5 inches.

Floppy disk drive - The electromechanical device that reads and writes floppy disks.

Track - Concentric ring of data on a side of a disk.

Sector - A subset of a track, similar to wedge or a slice of pie.

It consists of a read/write head and a motor rotating the disk at a high speed of about 300 rotations per minute. It can be fitted inside the cabinet of the computer and from outside, the slit where the disk is to be inserted, is visible. When the disk drive is closed after inserting the floppy inside, the monitor catches the disk through the Central of Disk hub, and then it starts rotating.

There are two read/write heads depending upon the floppy being one sided or two sided. The head consists of a read/write coil wound on a ring of magnetic material. During write operation, when the current passes in one direction, through the coil, the disk surface touching the head is magnetized in one direction. For reading the data, the procedure is reverse. I.e. the magnetized spots on the disk touching the read/write head induce the electronic pulses, which are sent to CPU.

The major parts of a FDD include:

Read/Write Heads: Located on both sides of a diskette, they move together on the same assembly. The heads are not directly opposite each other in an effort to prevent interaction between write operations on each of the two media surfaces. The same head is used for reading and writing, while a second, wider head is used for erasing a track just prior to it being written. This allows the data to be written on a wider "clean slate," without interfering with the analog data on an adjacent track.

Drive Motor: A very small spindle motor engages the metal hub at the center of the diskette, spinning it at either 300 or 360 rotations per minute (RPM).

Stepper Motor: This motor makes a precise number of stepped revolutions to move the read/write head assembly to the proper track position. The read/write head assembly is fastened to the stepper motor shaft.

Mechanical Frame: A system of levers that opens the little protective window on the diskette to allow the read/write heads to touch the dual-sided diskette media. An external button allows the diskette to be ejected, at which point the spring-loaded protective window on the diskette closes.

Circuit Board: Contains all of the electronics to handle the data read from or written to the diskette. It also controls the stepper-motor control circuits used to move the read/write heads to each track, as well as the movement of the read/write heads toward the diskette surface.

Electronic optics check for the presence of an opening in the lower corner of a 3.5-inch diskette (or a notch in the side of a 5.25-inch diskette) to see if the user wants to prevent data from being written on it.

Hard Disks

Your computer uses two types of memory: primary memory which is stored on chips located on the motherboard, and secondary memory that is stored in the hard drive. Primary memory holds all of the essential memory that tells your computer how to be a computer. Secondary memory holds the information that you store in the computer.

Inside the hard disk drive case you will find circular disks that are made from polished steel. On the disks, there are many tracks or cylinders. Within the hard drive, an electronic reading/writing device called the head passes back and forth over the cylinders, reading information from the disk or writing information to it. Hard drives spin at 3600 or more rpm (Revolutions Per Minute) - that means that in one minute, the hard drive spins around over 7200 times!

Optical Storage

- Compact Disk Read-Only Memory (CD-ROM)
- CD-Recordable (CD-R)/CD-Rewritable (CD-RW)
- Digital Video Disk Read-Only Memory (DVD-ROM)
- DVD Recordable (DVD-R/DVD Rewritable (DVD-RW)
- Photo CD

Optical Storage Devices Data is stored on a reflective surface so it can be read by a beam of laser light. Two Kinds of Optical Storage Devices

- CD-ROM (compact disk read-only memory)
- DVD-ROM (digital video disk read-only memory)

Compact Disks

Instead of electromagnetism, CDs use pits (microscopic indentations) and lands (flat surfaces) to store information much the same way floppies and hard disks use magnetic and non-magnetic storage. Inside the CD-Rom is a laser that reflects light off of the surface of the disk to an electric eye. The pattern of reflected light (pit) and no reflected light (land) creates a code that represents data.

CDs usually store about 650MB. This is quite a bit more than the 1.44MB that a floppy disk stores. A DVD or Digital Video Disk holds even more information than a CD, because the DVD can store information on two levels, in smaller pits or sometimes on both sides.

Recordable Optical Technologies

- CD-Recordable (CD-R)
- CD-Rewritable (CD-RW)

- PhotoCD
- DVD-Recordable (DVD-R)
- DVD-RAM

CD ROM - Compact Disc Read Only Memory.

Unlike magnetic storage device which store data on multiple concentric tracks, all CD formats store data on one physical track, which spirals continuously from the center to the outer edge of the recording area. Data resides on the thin aluminum substrate immediately beneath the label. The data on the CD is recorded as a series of microscopic pits and lands physically embossed on an aluminum substrate. Optical drives use a low power laser to read data from those discs without physical contact between the head and the disc which contributes to the high reliability and permanence of storage device.

To write the data on a CD a higher power laser are used to record the data on a CD. It creates the pits and land on aluminum substrate. The data is stored permanently on the disc. These types of discs are called as WORM (Write Once Read Many). Data written to CD cannot subsequently be deleted or overwritten which can be classified as advantage or disadvantage depending upon the requirement of the user. However if the CD is partially filled then the more data can be added to it later on till it is full. CDs are usually cheap and cost effective in terms of storage capacity and transferring the data.

The CD's were further developed where the data could be deleted and re written. These types of CDs are called as CD Rewritable. These types of discs can be used by deleting the data and making the space for new data. These CD's can be written and rewritten at least 1000 times.

CD ROM Drive

CD ROM drives are so well standardized and have become so ubiquitous that many treat them as commodity items. Although CD ROM drives differ in reliability, which standards they support and numerous other respects, there are two important performance measures.

- Data transfer rate
- Average access

Data transfer rate: Data transfer rate means how fast the drive delivers sequential data to the interface. This rate is determined by drive rotation speed, and is rated by a number followed by 'X'. All the other things equal, a 32X drive delivers data twice the speed of a 16X drive. Fast data transfer rate is most important when the drive is used to transfer the large file or many sequential smaller files. For example: Gaming video.

CD ROM drive transfers the data at some integer multiple of this basic 150 KB/s 1X rate. Rather than designating drives by actual KB/s output drive manufacturers use a multiple of the standard 1X rate. For example: a 12X drive transfer data at (12*150KB/s) 1800 KB/s and so on.

The data on a CD is saved on tracks, which spirals from the center of the CD to outer edge. The portions of the tracks towards center are shorter than those towards the edge. Moving the data

under the head at a constant rate requires spinning the disc faster as the head moves from the center where there is less data per revolution to the edge where there is more data. Hence the rotation rate of the disc changes as it progresses from inner to outer portions of the disc.

CD Writers

CD recordable and CD rewritable drives are collectively called as CD writers or CD burners. They are essentially CD ROM drives with one difference. They have a more powerful laser that, in addition to reading discs, can record data to special CD media.

Pen Drives / Flash Drives

- Pen Drives / Flash Drives are flash memory storage devices.
- They are faster, portable and have a capability of storing large data.
- It consists of a small printed circuit board with a LED encased in a robust plastic
- The male type connector is used to connect to the host PC
- They are also used a MP3 players

NUMBER SYSTEMS

Binary	Decimal	Octal	Hexadecimal
0000	00	0	0
0001	01	1	1
0010	02	2	2
0011	03	3	3
0100	04	4	4
0101	05	5	5
0110	06	6	6
0111	07	7	7
1000	08	10	8
1001	09	11	9
1010	10	12	A
1011	11	13	B
1100	12	14	C
1101	13	15	D
1110	14	16	E
1111	15	17	F

DECIMAL NUMBERS

In the decimal number systems each of the ten digits, 0 through 9, represents a certain quantity. The position of each digit in a decimal number indicates the magnitude of the quantity represented and can be assigned a weight. The weights for whole numbers are positive powers of ten that increases from right to left, beginning with $10^o = 1$ that is $10^3\ 10^2\ 10^1\ 10^o$

For fractional numbers, the weights are negative powers of ten that decrease from left to right beginningwith 10^{-1} that is $10^2\ 10^1\ 10^o.\ 10^{-1}\ 10^{-2}\ 10^{-3}$
The value of a decimal number is the sum of digits after each digit has been multiplied by its weights asin following examples

Express the decimal number 87 as a sum of the values of each digit.
The digit 8 has a weight of10 which is 10 as indicated by its position. The digit 7 has a weight of 1 which is 10^o as indicated by its position.

$$\mathbf{87 = (8 \times 10^{1)} + (7 \times 10^0)}$$

Express the decimal number 725.45 as a sum of the values of each digit.

$$\mathbf{725.\ 45 = (7 \times 10^2) + (2 \times 10^1) + (5 \times 10^o) + (4 \times 10^{-1}) + (5 \times 10^{-2}) = 700 + 20 + 5 + 0.4 + 0.05}$$

BINARY NUMBERS

The binary system is less complicated than the decimal system because it has only two digits, it is a basetwo system. The two binary digits (bits) are 1 and 0. The position of a 1 or 0 in a binary number indicates its weight, or value within the number, just as the position of a decimal digit determines the value of that digit. The weights in a binary number are based on power of two as:

$$\mathbf{\ldots..\ 2^4\ 2^3\ 2^2\ 2^1\ 2^0.\ 2^{-1}\ 2^{-2}\ \ldots.}$$

With 4 digits position we can count from zero to 15.In general, with n bits we can count up to a number equal to Ķ - 1. Largest decimal number = Ķ - 1.A binary number is a weighted number. The right-most bit is the least significant bit (LSB) in a binary whole number and has a weight of $2^o = 1$. The weights increase from right to left by a power of two for each bit. The left-most bit is the most significant bit (MSB); its weight depends on the size of the binary number.

BINARY-TO-DECIMAL CONVERSION

The decimal value of any binary number can be found by adding the weights of all bits that are 1 and discarding the weights of all bits that are 0
Example
Let's convert the binary whole number 101101 to decimal

Weight:2^5 2^4 2^3 2^2 2^1 2^0
x

Binary no:	**1**	**0**	**1**	**1**	**0**	**1**
Value	**32**	**0**	**8**	**4**	**0**	**1**

Sum = 45

HEXADECIMAL NUMBERS

The hexadecimal number system has sixteen digits and is used primarily as a compact way of displaying or writing binary numbers because it is very easy to convert between binary and hexadecimal.Long binary numbers are difficult to read and write because it is easy to drop or transpose a bit. Hexadecimal is widely used in computer and microprocessor applications. The hexadecimal system has a base of sixteen; it is composed of 16 digits and alphabetic characters. The maximum 3-digits hexadecimal number is FFF or decimal 4095 and maximum 4-digit hexadecimal number is FFFF or decimal 65.535.

BINARY-TO-HEXADECIMAL CONVERSION

Simply break the binary number into 4-bit groups, starting at the right-most bit and replace each 4-bit group with the equivalent hexadecimal symbol as in the following example

Convert the binary number to hexadecimal: 1100101001010111
Solution:
1100 1010 0101 0111

C A 5 7 = CA57

HEXADECIMAL-TO-DECIMAL CONVERSION

One way to find the decimal equivalent of a hexadecimal number is to first convert the hexadecimal number to binary and then convert from binary to decimal.

Convert the hexadecimal number 1C to decimal:

1 C
0001 1100= $2^4 + 2^3 + 2^2$ = 16 +8+4 = 28

DECIMAL-TO-HEXADECIMAL CONVERSION

Repeated division of a decimal number by 16 will produce the equivalent hexadecimal number, formed by the remainders of the divisions. The first remainder produced is the least significant digit (LSD). Each successive division by 16 yields a remainder that becomes a digit in the equivalent hexadecimal number. When a quotient has a fractional part, the fractional part is multiplied by the divisor to get the remainder.

Convert the decimal number 650 to hexadecimal by repeated division by 16

650 /16 = 40.625 **0.625 x 16 = 10 = A (LSD)**
40 /16= 2.5 **0.5 x 16 = 8 = 8**
2/16= 0.125 **0.125 x 16 = 2 = 2 (MSD)**
The hexadecimal number is 28A

OCTAL NUMBERS

Like the hexadecimal system, the octal system provides a convenient way to express binary numbers and codes. However, it is used less frequently than hexadecimal in conjunction with computers and microprocessors to express binary quantities for input and output purposes.

The octal system is composed of eight digits, which are: 0, 1, 2, 3, 4, 5, 6, 7
To count above 7, begin another column and start over: 10, 11, 12, 13, 14, 15, 16, 17, 20, 21 and so on. Counting in octal is similar to counting in decimal, except that the digits 8 and 9 are not used.

OCTAL-TO-DECIMAL CONVERSION

Since the octal number system has a base of eight, each successive digit position is an increasing power of eight, beginning in the right-most column with 8^o. The evaluation of an octal number in terms of its decimal equivalent is accomplished by multiplying each digit by its weight and summing the products.

Let's convert octal number 2374 in decimal number.

Weight 8^3 8^2 8^1 8^0
Octal number 2 3 7 4
$2374 = (2 \times 8^3) + (3 \times 8^2) + (7 \times 8^1) + (4 \times 8^o) = 1276$

DECIMAL-TO-OCTAL CONVERSION

A method of converting a decimal number to an octal number is the repeated division-by-8 method, which is similar to the method used in the conversion of decimal numbers to binary or to hexadecimal.
Let's convert the decimal number 359 to octal. Each successive division by 8 yields a remainder that becomes a digit in the equivalent octal number. The first remainder generated is the least significant digit (LSD).

359/8 = 44.875 **0.875 x 8 = 7 (LSD)**
44 /8 = 5.5 **0.5 x 8 = 4**
5/8= 0.625 **0.625 x 8 = 5 (MSD)**
The number is 547.

OCTAL-TO-BINARY CONVERSION

Because each octal digit can be represented by a 3-bit binary number, it is very easy to convert from octal to binary.

Octal/Binary Conversion

Octal Digit	0	1	2	3	4	5	6	7
Binary	000	001	010	011	100	101	110	111

Let's convert the octal numbers 25 and 140.

Octal Digit	2	5		1	4	0
Binary	010	101		001	100	000

BINARY-TO-OCTAL CONVERSION

Conversion of a binary number to an octal number is the reverse of the octal-to-binary conversion.

Let's convert the following binary numbers to octal:

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

6 5 = 65 5 7 1 = 571

ALGORITHM

Algorithm

- Set of step-by-step instructions that perform a specific task or operation
- "Natural" language NOT programming language

Pseudocode

- Set of instructions that mimic programming language instructions

Flowchart

- Visual program design tool
- "Semantic" symbols describe operations to be performed

FLOWCHARTS

Definitions:

A flowchart is a schematic representation of an algorithm or a stepwise process, showing the steps as boxes of various kinds, and their order by connecting these with arrows. Flowcharts are used in designing or documenting a process or program.

A flow chart, or flow diagram, is a graphical representation of a process or system that details the sequencing of steps required to create output.

A flowchart is a picture of the separate steps of a process in sequential order.

TYPES:

High-Level Flowchart

A high-level (also called first-level or top-down) flowchart shows the major steps in a process. It illustrates a "birds-eye view" of a process, such as the example in the figure entitled High-Level Flowchart of Prenatal Care. It can also include the intermediate outputs of each step (the product or service produced), and the sub-steps involved. Such a flowchart offers a basic picture of the process and identifies the changes taking place within the process. It is significantly useful for identifying appropriate team members (those who are involved in the process) and for developing indicators for monitoring the process because of its focus on intermediate outputs.

Most processes can be adequately portrayed in four or five boxes that represent the major steps or activities of the process. In fact, it is a good idea to use only a few boxes, because doing so forces one to consider the most important steps. Other steps are usually sub-steps of the more important ones.

Detailed Flowchart

The detailed flowchart provides a detailed picture of a process by mapping all of the steps and activities that occur in the process. This type of flowchart indicates the steps or activities of a process and includes such things as decision points, waiting periods, tasks that frequently must be redone (rework), and feedback loops. This type of flowchart is useful for examining areas of the process in detail and for looking for problems or areas of inefficiency. For example, the Detailed Flowchart of Patient Registration reveals the delays that result when the record clerk and clinical officer are not available to assist clients.

Deployment or Matrix Flowchart

A deployment flowchart maps out the process in terms of who is doing the steps. It is in the form of a matrix, showing the various participants and the flow of steps among these participants. It is chiefly useful in identifying who is providing inputs or services to whom, as well as areas where different people may be needlessly doing the same task. See the Deployment of Matrix Flowchart.

ADVANTAGES OF USING FLOWCHARTS

The benefits of flowcharts are as follows:

1. **Communication**: Flowcharts are better way of communicating the logic of a system to all concerned.
2. **Effective analysis**: With the help of flowchart, problem can be analysed in more effective way.
3. **Proper documentation**: Program flowcharts serve as a good program documentation, which is needed for various purposes.

4. **Efficient Coding**: The flowcharts act as a guide or blueprint during the systems analysis and program development phase.
5. **Proper Debugging**: The flowchart helps in debugging process.
6. **Efficient Program Maintenance**: The maintenance of operating program becomes easy with the help of flowchart. It helps the programmer to put efforts more efficiently on that part

Advantages:

- Logic Flowcharts are easy to understand.They provide a graphical representation of actions to be taken.

- Logic Flowcharts are well suited for representing logic where there is intermingling among many actions.

Disadvantages:

- Logic Flowcharts may encourage the use of GoTo statements leadingsoftware design that is unstructured with logic that is difficult to decipher.

- Without an automated tool, it is time-consuming to maintain Logic Flowcharts.

- Logic Flowcharts may be used during detailed logic design to specify a module.

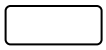
- However, the presence of decision boxes may encourage the use of GoTo statements, resulting in software that is not structured. For this reason, Logic Flowcharts may be better used during Structural Design

LIMITATIONS OF USING FLOWCHARTS

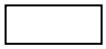
1. **Complex logic**: Sometimes, the program logic is quite complicated. In that case, flowchart becomes complex and clumsy.
2. **Alterations and Modifications**: If alterations are required the flowchart may require re-drawing completely.
3. **Reproduction**: As the flowchart symbols cannot be typed, reproduction of flowchart becomes a problem.
4. The essentials of what is done can easily be lost in the technical details of how it is done.

GUIDELINES FOR DRAWING A FLOWCHART

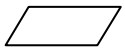
Flowcharts are usually drawn using some standard symbols; however, some special symbols can also be developed when required. Some standard symbols, which are frequently required for flowcharting many computer programs.



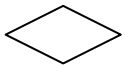
Start or end of the program



Computational steps or processing function of a program



Input or output operation



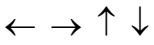
Decision making and branching

Connector or joining of two parts of program

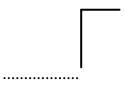
Magnetic Tape

Magnetic Disk

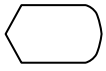
Off-page connector



Flow line



Annotation



Display

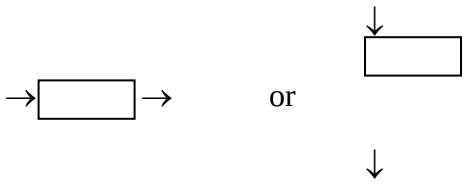
The following are some guidelines in flowcharting:

(a) In drawing a proper flowchart, all necessary requirements should be listed out in logical order.

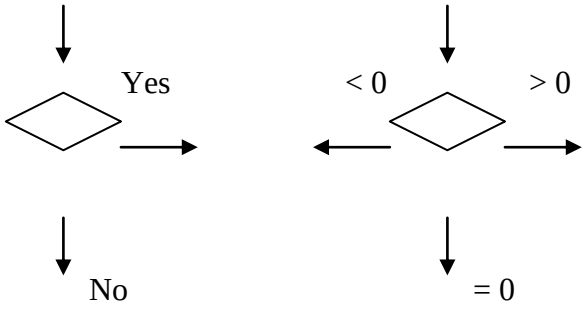
(b) The flowchart should be clear, neat and easy to follow. There should not be any room for ambiguity in understanding the flowchart.

(c) The usual direction of the flow of a procedure or system is from left to right or top to bottom.

(d) Only one flow line should come out from a process symbol.

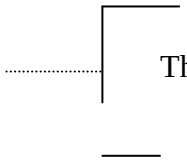


(e) Only one flow line should enter a decision symbol, but two or three flow lines, one for each possible answer, should leave the decision symbol.



(f) Only one flow line is used in conjunction with terminal symbol.

(g) Write within standard symbols briefly. As necessary, you can use the annotation symbol to describe data or computational steps more clearly.



(h) If the flowchart becomes complex, it is better to use connector symbols to reduce the number of flow lines. Avoid the intersection of flow lines if you want to make it more effective and better way of communication.

(i) Ensure that the flowchart has a logical *start* and *finish.*

(j) It is useful to test the validity of the flowchart by passing through it with a simple test data.

Examples

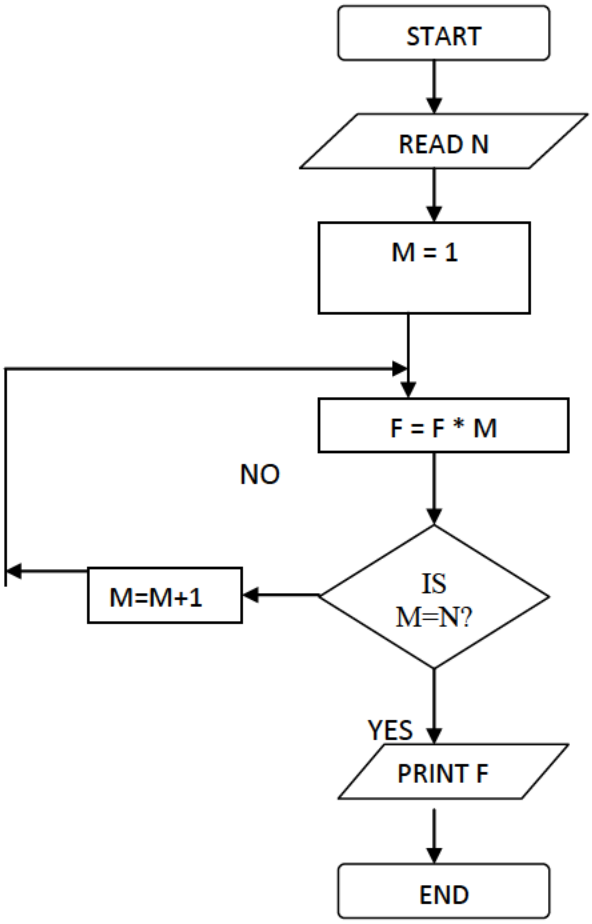
Sample flowchart

A flowchart for computing factorial N (N!) Where N! = 1 * 2 * 3 *...* N. This flowchart represents a "loop and a half" — a situation discussed in introductory programming textbooks that requires either a duplication of a component (to be both inside and outside the loop) or the component to be put inside a branch in the loop

Sample Pseudocode

```
ALGORITHM Sample
        GET Data
        WHILE There Is Data
                DO Math Operation
                GET Data
        END WHILE
END ALGORITHM
```

Flowchart for computing factorial N



UNIT II

C PROGRAMMING BASICS

Problem formulation – Problem Solving - Introduction to ‘ C’ programming –fundamentals – structureof a ‘C’ program – compilation and linking processes – Constants, Variables – Data Types –Expressions using operators in ‘C’ – Managing Input and Output operations – Decision Making and Branching – Looping statements – solving simple scientific and statistical problems.

INTRODUCTION TO C

As a programming language, C is rather like Pascal or Fortran.. Values are stored in variables. Programs are structured by defining and calling functions. Program flow is controlled using loops, if statements and function calls. Input and output can be directed to the terminal or to files. Related data can be stored together in arrays or structures.

Of the three languages, C allows the most precise control of input and output. C is also rather more terse than Fortran or Pascal. This can result in short efficient programs, where the programmer has made wise use of C's range of powerful operators. It also allows the programmer to produce programs which are impossible to understand. Programmers who are familiar with the use of pointers (or indirect addressing, to use the correct term) will welcome the ease of use compared with some other languages. Undisciplined use of pointers can lead to errors which are very hard to trace. This course only deals with the simplest applications of pointers.

A Simple Program

The following program is written in the C programming language.

```
#include <stdio.h>
main()
{
        printf("Programming in C is easy.\n");

}
```

BASIC STRUCTURE OF C PROGRAMS

C programs are essentially constructed in the following manner, as a number of well defined sections.

```
/* HEADER SECTION                                    */
/* Contains name, author, revision number*/

/* INCLUDE SECTION                                   */
/* contains #include statements                      */

/* CONSTANTS AND TYPES SECTION                                */
/* contains types and #defines
                                                     */
/* GLOBAL VARIABLES SECTION                          */
/* any global variables declared here
                                                     */
/* FUNCTIONS SECTION                                 */
/* user defined functions
                                                   */
/* main() SECTION                                    */


intmain()
{

}
```

CONSTANTS

A constant is an entity that doesn't change whereas a variable is an entity that may change.

Types of C Constants

C constants can be divided into two major categories:

(a) Primary Constants

(b) Secondary Constants

Rules for Constructing Integer Constants

(a) An integer constant must have at least one digit.
(b) It must not have a decimal point.
(c) It can be either positive or negative.
(d) If no sign precedes an integer constant it is assumed to be positive.
(e) No commas or blanks are allowed within an integer constant.
(f) The allowable range for integer constants is -32768 to 32767.

Truly speaking the range of an Integer constant depends upon the compiler. For a 16-bit compiler like Turbo C or Turbo C++ the range is –32768 to 32767. For a 32-bit compiler the range would be even greater. Question like what exactly do you mean by a 16-bit or a 32-bit compiler, what range of an Integer constant has to do with the type of compiler and such questions are discussed in detail in Chapter 16. Till that time it would be assumed that we are working with a 16-bit compiler.
Ex.: 426
+782
-8000
-7605

Rules for Constructing Real Constants

Real constants are often called Floating Point constants. The real constants could be written in two forms—Fractional form and Exponential form.
Following rules must be observed while constructing real constants expressed in fractional form:
(a)A real constant must have at least one digit.
(b)It must have a decimal point.
(c)It could be either positive or negative.
(d)Default sign is positive.
(e)No commas or blanks are allowed within a real constant.
Ex.: +325.34
426.0
-32.76
-48.5792

The exponential form of representation of real constants is usually used if the value of the constant is either too small or too large. It however doesn't restrict us in any way from using exponential form of representation for other real constants.

Rules for Constructing Character Constants

A character constant is a single alphabet, a single digit or a single special symbol enclosed within single inverted commas. Both the inverted commas should point to the left. For example, 'A' is a valid character constant whereas 'A' is not.
The maximum length of a character constant can be 1 character.
Ex.: 'A'
'I'
'5'
'='

VARIABLES

User defined variables must be declared before they can be used in a program. Variables must begin with a character or underscore, and may be followed by any combination of characters, underscores, or the digits 0 - 9.

LOCAL AND GLOBAL VARIABLES

Local
These variables only exist inside the specific function that creates them. They are unknown to other functions and to the main program. As such, they are normally implemented using a stack. Local variables cease to exist once the function that created them is completed. They are recreated each time a function is executed or called.

Global

These variables can be accessed (ie known) by any function comprising the program. They are implemented by associating memory locations with variable names. They do not get recreated if the function is recalled.

Defining Global Variables

/* Demonstrating Global variables */

Example:

```
#include <stdio.h>
int add_numbers( void );                    /* ANSI function prototype */

/* These are global variables and can be accessed by functions from this point on */
 int  value1, value2, value3;
int add_numbers( void )
{
auto int result;
result = value1 + value2 + value3;
return result;
}
main()
{
auto int result;
result = add_numbers();
printf("The sum of %d + %d + %d is %d\n", value1, value2, value3, final_result);
            }
```

The scope of global variables can be restricted by carefully placing the declaration. They are visible from the declaration until the end of the current source file.

Example:

```
#include <stdio.h>
void no_access( void ); /* ANSI function prototype */ void all_access( void );
static int n2;/* n2 is known from this point onwards */
void no_access( void )
{
n1 = 10; /* illegal, n1 not yet known */
n2 = 5; /* valid */
}
static int n1; /* n1 is known from this point onwards */
void all_access( void )
{
n1 = 10;                                    /* valid */
```

```
n2 = 3;                                 /* valid */
}
```

AUTOMATIC AND STATIC VARIABLES

C programs have a number of segments (or areas) where data is located. These segments are typically, _DATA Static data _BSS Uninitialized static data, zeroed out before call to main() _STACK Automatic data, resides on stack frame, thus local to functions _CONST Constant data, using the ANSI C keyword const

The use of the appropriate keyword allows correct placement of the variable onto the desired data segment.

Example:

```
/* example program illustrates difference between static and automatic variables */
 #include <stdio.h>
void demo( void ); /* ANSI function prototypes */
void demo( void )
{
auto int avar = 0;  static int svar = 0;

printf("auto = %d, static = %d\n", avar, svar);
++avar; ++svar;
}

main()
{
int i;

while( i < 3 ) {
demo();
i++;
}
}
```

Automatic and Static Variables

Example:

```
/* example program illustrates difference between static and automatic variables */
#include <stdio.h>
void demo( void ); /* ANSI function prototypes */

void demo( void )  {
auto int avar = 0;  static int svar = 0;
printf("auto = %d, static = %d\n", avar, svar);
++avar; ++svar;
```

```
}

main()
{
int i;

while( i < 3 ) {
demo();
i++;
}
}
```

Program output

```
auto =  0,  static =  0
auto =  0,  static =  1
auto = 0, static = 2
```

The basic format for declaring variables is

data_type var, var, ... ;
where data_type is one of the four basic types, an integer, character, float, or double type.

Static variables are created and initialized once, on the first call to the function. Subsequent calls to the function do not recreate or re-initialize the static variable. When the function terminates, the variable still exists on the _DATA segment, but cannot be accessed by outside functions. Automatic variables are the opposite. They are created and re-initialized on each entry to the function. They disappear (are de-allocated) when the function terminates. They are created on the _STACK segment.

DATA TYPES

The four basic data types are

INTEGER

These are whole numbers, both positive and negative. Unsigned integers (positive values only) are supported. In addition, there are short and long integers.

The keyword used to define integers is,

int
An example of an integer value is 32. An example of declaring an integer variable called sum is,
int sum;
sum = 20;

FLOATING POINT

These are numbers which contain fractional parts, both positive and negative. The keyword used to define float variables is,

float

An example of a float value is 34.12. An example of declaring a float variable called money is,

```
float money;
money = 0.12;
```

DOUBLE

These are exponetional numbers, both positive and negative. The keyword used to define double variables
is,

double

An example of a double value is 3.0E2. An example of declaring a double variable called big is,

```
double big;
big = 312E+7;
```

CHARACTER

These are single characters. The keyword used to define character variables is,

char

An example of a character value is the letter A. An example of declaring a character variable called letter
is,

```
Char letter;
letter = 'A';
```

Sample program illustrating each data type

Example:

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

```
{
int   sum;
float money;
char  letter;
double pi;

sum = 10;                                   /* assign integer value */
money = 2.21;                                       /* assign float value */
letter = 'A';                                       /* assign character value */
pi = 2.01E6;                                        /* assign a double value */
printf("value of sum = %d\n", sum );
            printf("value of money = %f\n", money );
            printf("value of letter = %c\n", letter );
            printf("value of pi = %e\n", pi );
             }
```

Sample program output
value of sum = 10
value of money = 2.210000
value of letter = A
value of pi = 2.010000e+06

INITIALISING DATA VARIABLES AT DECLARATION TIME

In C variables may be initialised with a value when they are declared. Consider the following declaration, which declares an integer variable count which is initialised to 10.

```
int  count = 10;
```

SIMPLE ASSIGNMENT OF VALUES TO VARIABLES

The = operator is used to assign values to data variables. Consider the following statement, which assigns the value 32 an integer variable count, and the letter A to the character variable letter

```
count = 32;
letter = 'A'
```

Variable Formatters

%d	decimal integer
%c	character
%s	string or character array
%f	float

%e double

HEADER FILES

Header files contain definitions of functions and variables which can be incorporated into any C program by using the pre-processor #include statement. Standard header files are provided with each compiler, and cover a range of areas, string handling, mathematical, data conversion, printing and reading of variables.
To use any of the standard functions, the appropriate header file should be included. This is done at the beginning of the C source file. For example, to use the function printf() in a program, the line

#include <stdio.h>
should be at the beginning of the source file, because the definition for printf() is found in the file stdio.h All header files have the extension .h and generally reside in the /include subdirectory.

#include <stdio.h>
#include "mydecls.h"
The use of angle brackets <> informs the compiler to search the compilers include directory for the specified file. The use of the double quotes "" around the filename inform the compiler to search in the current directory for the specified file.

OPERATORS AND EXPRESSIONS

An ex pr e s s i o n is a sequence of operators and operands that specifies computation of a value, or that designates an object or a function, or that generates side effects, or that performs a combination thereof.

ARITHMETIC OPERATORS:

The symbols of the arithmetic operators are:-

Operation	Operator	Comment	Value of Sum before	Value of sum after
Multiply	*	sum = sum * 2;	4	8
Divide	/	sum = sum / 2;	4	2
Addition	+	sum = sum + 2;	4	6
Subtraction	-	sum = sum -2;	4	2
Increment	++	++sum;	4	5
Decrement	--	--sum;	4	3
Modulus	%	sum = sum % 3;	4	1

Example:

```
#include <stdio.h>
main()
{
int sum = 50; float modulus;

modulus = sum % 10;
        printf("The %% of %d by 10 is %f\n", sum, modulus);
}
```

PRE/POST INCREMENT/DECREMENT OPERATORS

PRE means do the operation first followed by any assignment operation. POST means do the operation after any assignment operation. Consider the following statements

```
++count;                    /* PRE Increment, means add one to count */
count++;                    /* POST Increment, means add one to count */
```

Example:

```
#include <stdio.h>
main()
{
int count = 0, loop;
loop = ++count;   /* same as count = count + 1; loop = count; */
printf("loop = %d, count = %d\n", loop, count);
loop = count++;   /* same as loop = count;  count = count + 1;   */
printf("loop = %d, count = %d\n", loop, count);
}
```

If the operator precedes (is on the left hand side) of the variable, the operation is performed first, so the statement loop = ++count;
really means increment count first, then assign the new value of count to loop.

THE RELATIONAL OPERATORS

These allow the comparison of two or more variables.

```
==   equal to
!=   not equal
```

< less than
<= less than or equal to
> greater than
>= greater than or equal to

Example:

```
#include <stdio.h>
main() /* Program introduces the for statement, counts to ten */
{
int  count;
for( count = 1; count <= 10; count = count + 1 )
printf("%d ", count );
printf("\n");
}
```

RELATIONALS (AND, NOT, OR, EOR)

Combining more than one condition
These allow the testing of more than one condition as part of selection statements. The symbols are

LOGICAL AND &&
Logical and requires all conditions to evaluate as TRUE (non-zero).

LOGICAL OR ||
Logical or will be executed if any ONE of the conditions is TRUE (non-zero).

LOGICAL NOT !
logical not negates (changes from TRUE to FALSE, vsvs) a condition.

LOGICAL EOR ^
Logical eor will be excuted if either condition is TRUE, but NOT if they are all true.

Example:

The following program uses an if statement with logical AND to validate the users input to be in the range 1-10.

```
#include <stdio.h>
main()
{
int number;
int valid = 0;
while( valid == 0 )
```

```
{
printf("Enter a number between 1 and 10 -->"); scanf("%d", &number);
if( (number < 1 ) || (number > 10) )
{
printf("Number is outside range 1-10. Please re-enter\n"); valid = 0;
}
else
valid = 1;
            }
            printf("The number is %d\n", number );
}
```

Example:

NEGATION

```
#include <stdio.h>
main()
{
int flag = 0;
if( ! flag )
{
printf("The flag is not set.\n");
flag = ! flag;
}
printf("The value of flag is %d\n", flag);
}
```

Example:

Consider where a value is to be inputted from the user, and checked for validity to be within a certain range, lets say between the integer values 1 and 100.

```
#include <stdio.h>
main()
{
int number;
int valid = 0;
while( valid == 0 ) {
printf("Enter a number between 1 and 100"); scanf("%d", &number );
if( (number < 1) || (number > 100) )
            printf("Number is outside legal range\n");
            else
                                    valid = 1;
            }
```

```
        printf("Number is %d\n", number );
}
```

THE CONDITIONAL EXPRESSION OPERATOR or TERNARY OPERATOR

This conditional expression operator takes THREE operators. The two symbols used to denote this operator are the ? and the :. The first operand is placed before the ?, the second operand between the ? and the :, and the third after the :. The general format is,

condition ? expression1 : expression2

If the result of condition is TRUE (non-zero), expression1 is evaluated and the result of the evaluation becomes the result of the operation. If the condition is FALSE (zero), then expression2 is evaluated and its result becomes the result of the operation. An example will help,

s = (x < 0) ? -1 : x * x;

If x is less than zero then s = -1
If x is greater than zero then s = x * x

Example:

```
#include <stdio.h>
main()
{
int input;
printf("I will tell you if the number is positive, negative or zero!"\n");
printf("please enter your number now--->");
scanf("%d", &input );
(input < 0) ? printf("negative\n") : ((input > 0) ? printf("positive\n") : printf("zero\n"));


}
```

BIT OPERATIONS

C has the advantage of direct bit manipulation and the operations available are,

Operation	Operator	Comment	Value of Sum before	Value of sum after
AND	&	sum = sum & 2;	4	0
OR	|	sum = sum | 2;	4	6
Exclusive OR	^	sum = sum ^ 2;	4	6
1's Complement	~	sum = ~sum;	4	-5
Left Shift	<<	sum = sum << 2;	4	16

Right Shift	>>	sum = sum >> 2;	4	0

Example:

```
/* Example program illustrating << and >> */

#include <stdio.h>
main()
{
int  n1 = 10, n2 = 20, i = 0;
i = n2 << 4;   /* n2 shifted left four times */
printf("%d\n", i);
i = n1 >> 5;   /* n1 shifted right five times */
printf("%d\n", i);
}
```

Example:

```
/* Example program using EOR operator   */
#include <stdio.h>
main()
{
int  value1 = 2, value2 = 4;
value1 ^= value2;
value2 ^= value1;
value1 ^= value2;
printf("Value1 = %d, Value2 = %d\n", value1, value2);
}
```

Example:

```
/* Example program using AND operator   */
#include <stdio.h>
main()
{
int  loop;
for( loop = 'A'; loop <= 'Z'; loop++ )
printf("Loop = %c, AND 0xdf = %c\n", loop, loop & 0xdf);
}
```

MANAGING INPUT AND OUTPUT OPERATORS

Printf ():

printf() is actually a function (procedure) in C that is used for printing variables and text. Where text appears in double quotes "", it is printed without modification. There are some exceptions however. This has to do with the \ and % characters. These characters are modifier's, and for the present the \ followed by the n character represents a newline character.

Example:

```
#include <stdio.h>
main()
{
        printf("Programming in C is easy.\n");
        printf("And so is Pascal.\n");
}
```

```
@ Programming in C is easy.
                And so is Pascal.
```

FORMATTERS for printf are,

Cursor Control Formatters

\n	newline
\t	tab
\r	carriage return
\f	form feed
\v	vertical tab

Scanf ():

Scanf () is a function in C which allows the programmer to accept input from a keyboard.

Example:

```
#include <stdio.h>

main()                          /* program which introduces keyboard input */
{
        int  number;

        printf("Type in a number \n");
        scanf("%d", &number);
        printf("The number you typed was %d\n", number);
}
```

FORMATTERS FOR scanf()

The following characters, after the % character, in a scanf argument, have the following effect.

d	read a decimal integer
o	read an octal value
x	read a hexadecimal value
h	read a short integer
l	read a long integer
f	read a float value
e	read a double value
c	read a single character
s	read a sequence of characters
[...]	Read a character string. The characters inside the brackets

Accepting Single Characters From The Keyboard

Getchar, Putchar

getchar() gets a single character from the keyboard, and **putchar()** writes a single character from the keyboard.

Example:

The following program illustrates this,

```
#include <stdio.h>
main()
{
int i;
int ch;
for( i = 1; i<= 5; ++i )
{
ch = getchar(); putchar(ch);
}
}
```

The program reads five characters (one for each iteration of the for loop) from the keyboard. Note that getchar() gets a single character from the keyboard, and putchar() writes a single character (in this case, ch) to the console screen.

DECISION MAKING

IF STATEMENTS

The if statements allows branching (decision making) depending upon the value or state of variables. This allows statements to be executed or skipped, depending upon decisions. The basic format is,

```
if( expression )
program statement;
```

Example:

```
if( students < 65 )
++student_count;
```

In the above example, the variable student_count is incremented by one only if the value of the integer variable students is less than 65.

The following program uses an if statement to validate the users input to be in the range 1-10.

Example:

```
#include <stdio.h>
main()
{
int number;
int valid = 0;
while( valid == 0 )
 {
printf("Enter a number between 1 and 10 -->");
scanf("%d", &number);
/* assume number is valid */ valid = 1;
if( number < 1 )
 {
printf("Number is below 1. Please re-enter\n"); valid = 0;
}
if( number > 10 )
{
printf("Number is above 10. Please re-enter\n"); valid = 0;
}
}
            printf("The number is %d\n", number );
}
```

IF ELSE

The general format for these are,

```
if( condition 1 )
statement1;
elseif(condition2)
statement2;
elseif(condition3)
statement3;
else
statement4;
```

The else clause allows action to be taken where the condition evaluates as false (zero).
The following program uses an if else statement to validate the users input to be in the range 1-10.

Example:

```
#include <stdio.h>
main()
{
int number;
int valid = 0;
while( valid == 0 )
 {
printf("Enter a number between 1 and 10 -->");
 scanf("%d", &number);
if( number < 1 )
{
printf("Number is below 1. Please re-enter\n"); valid = 0;
}
else if( number > 10 )
{
printf("Number is above 10. Please re-enter\n"); valid = 0;
}
else
valid = 1;
}
printf("The number is %d\n", number );
}
```

This program is slightly different from the previous example in that an else clause is used to set the

variable valid to 1. In this program, the logic should be easier to follow.

NESTED IF ELSE

/* Illustates nested if else and multiple arguments to the scanf function. */

Example:

```
#include <stdio.h>
main()
{
int  invalid_operator = 0;
char  operator;
float  number1, number2, result;
printf("Enter two numbers and an operator in the format\n");
printf(" number1 operator number2\n");
scanf("%f %c %f", &number1, &operator, &number2);
if(operator == '*')
result = number1 * number2;
elseif(operator=='/')
result=number1/number2;
elseif(operator=='+')
result=number1+number2;
elseif(operator=='-')
result=number1-number2;
else
invalid_operator = 1;
if( invalid_operator != 1 )
        printf("%f %c %f is %f\n", number1, operator, number2, result );
else
        printf("Invalid operator.\n");
}
```

BRANCHING AND LOOPING

ITERATION, FOR LOOPS

The basic format of the for statement is,

```
for(start condition; continue condition; re-evaulation )
program statement;
```

Example:

```
/*sample program using a for statement */
```

```
#include <stdio.h>

main()/* Program introduces the for statement, counts to ten */
{
int  count;
for( count = 1; count <= 10; count = count + 1 )
printf("%d ", count );
printf("\n");
}
```

The program declares an integer variable count. The first part of the for statement for(count = 1; initialises the value of count to 1. The for loop continues whilst the condition count <= 10; evaluates as TRUE. As the variable count has just been initialised to 1, this condition is TRUE and so the program statement printf("%d ", count); is executed, which prints the value of count to the screen, followed by a space character. Next, the remaining statement of the for is executed

count = count + 1);
which adds one to the current value of count. Control now passes back to the conditional test,
count <= 10;
which evaluates as true, so the program statement
printf("%d ", count);
is executed. Count is incremented again, the condition re-evaluated etc, until count reaches a value of 11.

When this occurs, the conditional test

count <= 10;
evaluates as FALSE, and the for loop terminates, and program control passes to the statement

printf("\n");
which prints a newline, and then the program terminates, as there are no more statements left to execute.

THE WHILE STATEMENT

The while provides a mechanism for repeating C statements whilst a condition is true. Its format is,

while(condition)
program statement;

Somewhere within the body of the while loop a statement must alter the value of the condition to allow the loop to finish.

Example:

```
/* Sample program including while */
#include <stdio.h>

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

The above program uses a while loop to repeat the statements

```
printf("%d\n", loop);
++loop;
```

whilst the value of the variable loop is less than or equal to 10.

Note how the variable upon which the while is dependant is initialised prior to the while statement (in this case the previous line), and also that the value of the variable is altered within the loop, so that eventually the conditional test will succeed and the while loop will terminate. This program is functionally equivalent to the earlier for program which counted to ten.

THE DO WHILE STATEMENT

The do { } while statement allows a loop to continue whilst a condition evaluates as TRUE (non-zero). The loop is executed as least once.

Example:

```
/* Demonstration of DO...WHILE*/

#include <stdio.h>
main()
{
int  value, r_digit;
printf("Enter the number to be reversed.\n");
scanf("%d", &value);
```

```
do
{
r_digit = value % 10;
printf("%d",r_digit);value=value/10;
 }
while(value!=0);
printf("\n");
}
```

The above program reverses a number that is entered by the user. It does this by using the modulus % operator to extract the right most digit into the variable r_digit. The original number is then divided by 10, and the operation repeated whilst the number is not equal to 0.

SWITCH CASE:

The switch case statement is a better way of writing a program when a series of if elses occurs. The general format for this is,

```
switch(expression)
{
case  value1:
Program statement;
program statement;
break;
case  valuen:
program statement;
break;
default:
break;
}
```

The keyword break must be included at the end of each case statement. The default clause is optional, and isexecuted if the cases are not met. The right brace at the end signifies the end of the case selections.

Example:

```
#include <stdio.h>
main()
{
int menu, numb1, numb2, total;
printf("enter in two numbers -->");
scanf("%d %d", &numb1, &numb2 );
printf("enter in choice\n");
printf("1=addition\n");
printf("2=subtraction\n");
```

```
scanf("%d",&menu);
switch( menu )
 {
case 1: total = numb1 + numb2; break;
case 2: total = numb1 - numb2; break;
        default:printf("Invalidoptionselected\n");
}
if( menu == 1 )
        printf("%d plus %d is %d\n", numb1, numb2, total ); else if( menu == 2 )
                                printf("%d minus %d is %d\n", numb1, numb2, total );
        }
```

The above program uses a switch statement to validate and select upon the users input choice, simulating a simple menu of choices.

UNIT III ARRAYS AND STRINGS

Arrays – Initialization – Declaration – One dimensional and Two dimensional arrays. String- String operations – String Arrays. Simple programs- sorting- searching – matrix operations.

3. ARRAYS

In the early chapters the values are stored in a variable of primary data type. The disadvantage is only one value can be stored in a variable at a time. Therefore to store more than one value, then more than one variable have to be declared. An additional variable can be declared to store or read two or three values. Imagine storing the register numbers of students of a college in several variables. If there are thousand students in a college then the program requires 1000 different variable names to store the register number of the students. Then the program becomes unreadable and complex. To simplify that array can be used by just declaring one variable of type array.

Consider the following example of data type int reg[1000], this means the variable name "reg" can store up to 1000 values. The values can be read, element by element. Therefore to store a large set of data of same type array is the choice. Array is a derived data type. It is very effective when working with large number of data of same data type or different data types. Array reduces the programming size or coding.

Definition:

Array is a collection of same data type elements under the same variable identifier referenced by index number. Arrays allow you to store group of data of a single type.

Consider the following example:

```
int x, y, z;
x=10; y=20; z=30;
```

The above statements can be rewritten using array form

```
int x[3];
x[0]=10; x[1]=20; x[2]=30;
```

When the number of variable is less the statement may not look confusing or difficult. Now consider the following example.

int y[60];

The array y can store up to 60 values in its elements. The structure and location value in the array is similar to matrix. Now it is clear that it is not good practice to declare 60 variables for array y.

Arrays are classified as follows:

1. One dimensional array
2. Two dimensional array
3. Multidimensional array
4. Dynamic array

3.1 One Dimensional Array:

It is similar to matrix with only one column but multiple rows. This type of array is useful when working with only one set of data at a time.

Declaring One Dimensional Array:

It is similar to declaring any other primary data type except, the number element must specified for an array in bracket [ ].

int z[5]; float s[10]; double w[6];
char t[4];

The arrays declared as shown. The numbers inside the bracket is the number of elements for that variable. Therefore z can store 5 values, s 10 values, w 6 values, and t 4 values. The only difference for character array t; is that the last element in the array must be allotted for NULL value, therefore remaining 3 elements of t can be any character including NULL.

Assigning One Dimensional Array:

int z[5]
z[0]=10;
z[1]=20;
z[2]=30;
z[3]=40;
z[4]=50;

The values are assigned element by element in the array.

Note: The array element start with 0 NOT 1

float s[10];
z[2]=11.11;
z[5]=22.22
z[8]=88.88;

In the above the element 2, 5, and 8 are assigned values but the other elements are not assigned any values. In such case the remaining element will be assigned zero. Therefore for numeric array. when the value is assigned to fewer elements the remaining elements will be assigned zero by the compiler.

char t[4];
t[0]='a';
t[1]='b';
t[2]='c';
t[3]='d';

The last element t[3] is assigned 'd', as we discussed earlier the last element for character array must be provided for NULL value. Here'd' is assigned to the last element the compiler will not give error during compilation but there will be error in the program during run time.

Now consider this example:

```
char t[4];
t[0]='a';
t[1]='b';
```

In this example the 3rd element and 4th element will be NULL. Therefore when the value is assigned to fewer elements the element will be assigned NULL by the compiler for character array whereas for numeric array zero will be assigned.

Initializing One Dimensional Array

Method 1:

```
int z[5] = {11,22,33,44,55};
char t[6] ={'a','p','p','l','e'};
```

All the array elements are assigned values. The last element for the character arrays must be provided for NULL character.

Method 2:

```
int z[5] = {11,22,33};
char t[6] ={'a','p','p'};
```

Only first three elements are assigned values the remaining elements will be assigned zero for integer data type. The remaining element for character array will be assigned NULL.

Method 3:

```
int z[5] = {0};
char t[6] = "apple";
```

All the array elements are assigned zero. The character array can also be initialized as above.

Method 4:

int z[ ] = {11,22,33};

char t[ ] = "apple";

Since array size is not defined. The array must be initialized, therefore the size of the array will be same the number of values assigned. Therefore the integer array size for this example is three since three values are provided in the initialization. The array size for character array t is 6, five for the character and one for the NULL terminator.

To assign zero to a specific element the location of the elements must be specified. Thus the array values are assigned sequentially. To assign value element by element then assigning method must be adopted.

z[5] = {0,22,33,44,55};

z[5] = {0,22,33,0,55};

Arrays can be read/written by any of the looping statements.

Assigning value to array by program:

```
int x[5],i;
for (i=0; i<5; i++)
{
 scanf("%d", &x[i]);
}
```

Reading value to array by program:

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

Remember, the initial value for array must start with zero and the condition part must be (<) less than sign, and the value must be the size of array, and the increment must by one, to assign value element by element.

Caution exceeding array boundary:

int x[3];

x[4]=10;

In this example a value has been assigned to the fifth element which exceeds the array size x, which is 3. The compiler will not give error during compilation. Therefore care must be taken to avoid exceeding the array boundary when assigning value to element.

3.2 Two Dimensional Arrays:

It is similar to matrix with multiple rows and columns. This type of array is useful when working with more than one set of data at a time.

Declaring Two Dimensional Arrays:

Syntax:

data_type var_name[r_s][c_s];

Eg: int marks[10][3];

The array has 10 rows and 3 columns. The values in the array are located by row and column, just like in matrix. Similar to one dimensional array the row and column elements start from zero not one.

Assigning values to Two Dimensional Arrays:

Consider the values in the following matrix.

	C0	C1	C2
R0	11	22	33
R1	44	0	55
R2	0	2	4

The above example has 3 rows and 3 columns. The value of the matrix can be assigned to the array as follows.

int rc[3][3];

rc[0][0]=11; rc[0][1]=22; rc[0][2]=33;

rc[1][0]=44; rc[1][1]=0; rc[1][2]=55;
rc[2][0]=0; rc[2][1]=2; rc[2][2]=4;

Initializing Two Dimensional Arrays:

Method 1:

Consider the following example for the matrix to be initialized:

```
int rc[3][3] = {11,22,33,44,0,55,0,2,4};
```

In this example the first row will be 11, 22, and 33; the second row 44, 0, and 55; and the third row 0, 2, and 4. Therefore it is clear that the values will be assigned in row wise.

Method 2:

The method 1 can be written as follows to get the same result.

```
int rc[3][3] = {{11,22,33},{44,0,55},{0,2,4}};
```

Here each braces { } represents each row, starting from the first row. The method two can bewritten to look like matrix without any change in the syntax:

```
int rc[3][3] = {{11,22,33},{44,0,55},{0,2,4}};
```

Method 3:

The method 1 can be written as follows to get the same result.

```
int rc[ ][3] = {{11,22,33},{44,0,55},{0,2,4}};
```

In this example the row will be set to 3 since row size is not specified the number of rows will be decided by the initial values.

Method 4:

```
int rc[3][3] = {{33},{0,55,0},{0}};
```

In this method the value in the array will be as follows

rc[0][0]=33; rc[0][1]=0; rc[0][2]=0;
rc[1][0]=0; rc[1][1]=55; rc[1][2]=0;
rc[2][0]=0; rc[2][1]=0; rc[2][2]=0;

Method 5:

```
int rc[3][3] = {{0},{0},{0}};
int rc[3][3] = {0,0,0};
```

In this method the value in the entire array element will be zero.

```
rc[0][0]=0; rc[0][1]=0; rc[0][2]=0;
rc[1][0]=0; rc[1][1]=0; rc[1][2]=0;
rc[2][0]=0; rc[2][1]=0; rc[2][2]=0;
```

Assigning value to array elements by program:

```
int rc[3][3],i,j;
for (i=0; i<3; ++i)
{
for (j=0;j<3;++j)
{
 scanf("%d", &rc[i][j]);
}
}
```


Reading value from array elements by program:

```
int rc[3][3],i,j;
for (i=0; i<3; ++i)
{
for (j=0;j<3;++j)
{
 printf("%d", rc[i][j]);
}
}
```

Remember for both row and column, the initial value for array must start with zero and the condition part must be less than sign (<) and the value must be the size of array, and the increment must by one, to assign value element by element.

Advantages of Array:

1. It reduces the programming length significantly.
2. It has better control over program for modification.
3. Coding is simple, efficient, and clear.
4. It can handle large set of data with simple declaration.

Rules of Array:

1. It does not check the boundary.
2. Processing time will increase when working with large data because of increased memory.
3. The array element start with zero not 1.
4. Character array size must be one element greater than data for NULL value.
5. One variable for control structure is required to assign and read value to one dimensional array.
6. Two variable for control structures are required to assign and read value to two dimensional arrays.
7. No two arrays can have the same name but arrays and ordinary variable can be assigned the same name.

3.3 STRING

String is the collection of characters. It can be represented by 1-d arrays. Header file used is **string.h**

String Declaration:

Datatype variable_name [size];

Eg: char sar[30];

String Initialization:

Datatype variable_name [size] = string;

Eg: char sar[30]={"Niro"};

3.3.1 Built-in String functions/ String Operations/ String Manipulation Functions:

There are several string functions to work with string variables and its values. These functions are available C header file called string.h. Consider the following example:

char string1[15]="Sriram";
char string2[15]="College";

- **Copying String**

  strcpy(string1,string2);

  This function copy's the value of string2 to string1. Now the string1 will be "College". To add the string2 to string1 the size of the string1 must be sufficient enough to fit the value of string2. This function will replace the existing value string1 with string2. Now string1 and string2 are "College".

- **String comparison [Case Sensitive]**

  strcmp(string1,string2);

  This function compares the value from string2 with string1. If both the string1 and string2 are exactly the same then the function will return zero or else it will return some positive or negative value. For the above example the function will return negative of positive value. Here string1 and string2 will not change.

  Non-Case Sensitive:

  strcmpi(string1, string2);

- **Concatenation String**

  strcat(string1,string2);

  This function is used to join two strings.

- **Copying String**

  srtcpy(string1,string2);

  This function copy's the contents of one string into another string.

- **Find a value in string**

  strstr(string1, string2);

  This function will find the value of string2 in string1. Assume string1 as "Apple" and string2 as "Ap", now the function will return position of first occurrence of "Ap", since "Ap" is found in "Apple".

- **Reversing a string**

  strrev(string1);

  This function reverse the data of string1 and stores it in string1.

- **Length of String**

  strlen(string1);

  This function will return length of the string. For the above example it returns 8.

- **Convert Uppercase to Lower case**

  strlwr(string1);

- **Convert Lower case to Uppercase**

  strupr(string1);

Example: Palindrome of string data

```
//To check whether the data is Palindrome
#include <stdio.h>
#include <conio.h>
#include <string.h>
int r;
char s1[15], s2[15];
void main()
{
clrscr();
printf("Enter anything :");
```

```
scanf("%s", s1);
strcpy(s2,s1);//copy's s1 to another variable s2
strrev(s2); //reverse the value of s2
printf("%s\n", s1);
printf("%s\n", s2);
r= strcmp(s1,s2);
if (r==0)
printf("It is a Palindrome %s\n", s1);
else
printf("It is not a Palindrome %s\n", s1);
getch();
}
```

EXPECTED PROGRAMS IN UNIT-3

1-D ARRAYS

P_1: Accept 5 numbers & store in Array also print the numbers in the Array.

```
#include <stdio.h>
#include <conio.h>
void main()
{
int a[5],i;
printf("\n Enter the elements:");
for(i=0;i<5;i++)
{
scanf("%d", &a[i]);
}
printf("\n The elements in the array are:");
for(i=0;i<5;i++)
{
printf("%d", a[i]);
```

```
printf("\n");
}
getch();
}
```

P_2: Reversing the elements in the Array.

```
#include <stdio.h>
#include <conio.h>
void main()
{
int a[10],i;
printf("\n Enter the elements:");
for(i=0;i<10;i++)
{
scanf("%d", &a[i]);
}
printf("\n The elements in the array are:");
for(i=9;i<=0;i--)
{
printf("%d", a[i]);
printf("\n");
}
getch();
}
```

P_3: To find the sum and average of elements in the Array.

```
#include <stdio.h>
#include <conio.h>
void main()
{
int a[20],i,sum=0,
float avg=0;
printf("\n Enter the elements:");
for(i=0;i<20;i++)
{
scanf("%d", &a[i]);
}
printf("\n The elements in the array are:");
for(i=0;i<20;i++)
{
sum=sum+a[i];
}
printf("\n The Sum of elements in the array is %d",sum);
avg=sum/20;
Printf("\n The Average of elements in the array is %f",avg);
getch();
}
```

P_4: Find the maximum of elements present in the array

```
#include <stdio.h>
#include <conio.h>
void main()
{
int i=0,x,a[5] ={32,44,11,3,6};
clrscr();
x=a[0];
for (i=1; i<5;i++)
{
 if (x<a[i])
x=a[i];
}
printf("\nThe value is %d.", x);
getch();
}
```


P_5: Program to arrange an array of elements in Ascending order. [BUBBLE SORT]

```
#include<stdio.h>
#include<conio.h>
void main()
{
int a[10],i,n,j,t;
clrscr();
printf("enter size of array: ");
scanf("%d",&n);
printf(" enter array elements: ");
for(i=0;i<n;i++)
{
```

```
scanf("%d",&a[i]);
}
printf(" array before sorting\n");
for(i=0;i<n;i++)
{
printf("%d\n",a[i]);
}
for(i=0;i<n;i++)
{
for(j=i+1;j<n;j++)
{
if(a[i]>a[j])
{
t=a[i];
a[i]=a[j];
a[j]=t;
}
}
}
printf("\n after sorting\n");
for(i=0;i<n;i++)
{
printf("%d\n",a[i]);
}
getch();
}
```

P_6: Program to arrange an array of elements in Ascending order. [QUICK SORT]

```
#include<stdio.h>
#include<conio.h>
void quicksort(int [10],int,int);
int main(){
  int x[20],size,i;
  printf("Enter size of the array: ");
  scanf("%d",&size);
  printf("Enter %d elements: ",size);
  for(i=0;i<size;i++)
    scanf("%d",&x[i]);
  quicksort(x,0,size-1);
  printf("Sorted elements: ");
  for(i=0;i<size;i++)
    printf(" %d",x[i]);
  return 0;
}
void quicksort(int x[10],int first,int last){
    int pivot,j,temp,i;
     if(first<last){
        pivot=first;
        i=first;
        j=last;
        while(i<j){
           while(x[i]<=x[pivot]&&i<last)
              i++;
           while(x[j]>x[pivot])
              j--;
           if(i<j){
              temp=x[i];
              x[i]=x[j];
              x[j]=temp;
```

```
        }
      }
      temp=x[pivot];
      x[pivot]=x[j];
      x[j]=temp;
      quicksort(x,first,j-1);
      quicksort(x,j+1,last);
   }
}
```

////**[2-Dimensional Arrays]**////

P_1: Initializing 2-D Array.

```
#include<stdio.h>
#include<conio.h>
void main()
{
int a[5][5],i,j;
printf("\n Enter the elements:");
for(i=0;i<5;i++)
{
for(j=0;j<5;j++)
{
printf("Matrix[%d][%d]",i,j);
scanf("%d", &a[i][j]);
}
}
printf("\n Matrix is:);
for(i=0;i<5;i++)
{
for(j=0;j<5;j++)
```

```
{
printf("%d", a[i]);
printf("\n");
}
}
getch();
}
```

P_2: Program to demonstrate matrix multiplication.

Algorithm-Multiplication of two matrixes:

Rule: Multiplication of two matrixes is only possible if first matrix has size m X **n** and other matrix has size **n** x r. Where m, n and r are any positive integer.

Multiplication of two matrixes is defined as

$$[AB]_{i,j} = \sum_{s=1}^{n} A_{i,s} B_{s,j}$$

Where $1 \le i \le m$ and $1 \le j \le n$

For example:

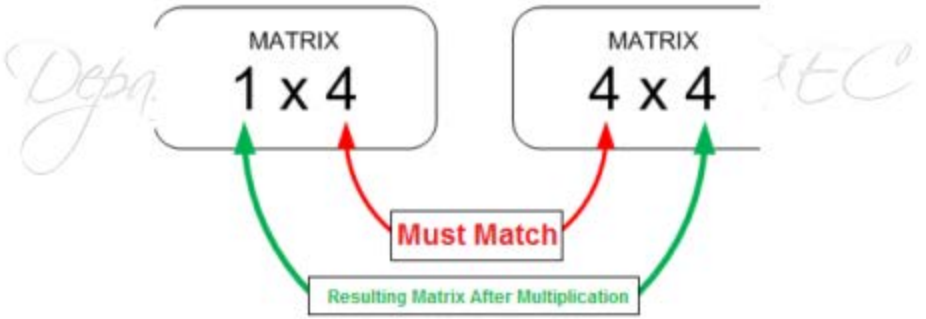
Suppose two matrixes A and B of size of 2 x 2 and 2 x 3 respectively:

$$A = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \quad B = \begin{pmatrix} 5 & 6 & 7 \\ 8 & 9 & 10 \end{pmatrix}$$

Multiplication of two matrixes:

$$A * B = \begin{pmatrix} 1*5 + 2*8 & 1*6 + 2*9 & 1*7 + 2*10 \\ 3*5 + 4*8 & 3*6 + 4*9 & 3*7 + 4*10 \end{pmatrix}$$

$$A * B = \begin{pmatrix} 21 & 24 & 27 \\ 47 & 54 & 61 \end{pmatrix}$$



Program:

```
#include<stdio.h>
#include<conio.h>
void main()
{
int a[10][10],b[10][10],c[10][10],i,j,k,row1,col1,row21,col2;
clrscr();
printf("Enter array1 size ");
```

```
scanf("%d%d",&row1,&col1);
printf("Enter array2 size ");
scanf("%d%d",&row2,&col2);
if(row2!=col1)
{
printf("Wrong choice entered");
getch();
exit(0);
}
else
{
printf("Enter elements");
for(i=0;i<row1;i++)
{
for(j=0;j<col1;j++)
{
scanf("%d",&a[i][j]);
}
}
printf("Enter element 2");
for(i=0;i<row2;i++)
{
for(j=0;j<col2;j++)
{
scanf("%d",&b[i][j]);
}
}
for(i=0;i<row1;i++)
{
for(j=0;j<col2;j++)
{
```

```
c[i][j]=0;
for(k=0;k<row2;k++)
{
c[i][j]=c[i][j]+(a[i][k]*b[k][j]);
}
}
}
printf("REQUIRED MATRIX");
for(i=0;i<row1;i++)
{
for(j=0;j<col2;j++)
{
printf("\n%d",c[i][j]);
}
printf("\n");
}
getch();
}
}
```


P_3: Program to demonstrate matrix addition.

```
#include<stdio.h>
#include<conio.h>
int main(){
  int a[3][3],b[3][3],c[3][3],i,j,row1,col1,row2,col2;
printf("Enter array1 size ");
scanf("%d%d",&row1,&col1);
printf("Enter array2 size ");
```

```
scanf("%d%d",&row2,&col2);

 printf("Enter the First matrix->");
 for(i=0;i<row1;i++)
{
    for(j=0;j<col1;j++)
{
        scanf("%d",&a[i][j]);
}
}
 printf("\nEnter the Second matrix->");
 for(i=0;i<row2;i++)
{
    for(j=0;j<col2;j++)
{
        scanf("%d",&b[i][j]);
}
}
  for(i=0;i<row1;i++)
{
   for(j=0;j<col1;j++)
{
         c[i][j]=a[i][j]+b[i][j];
}
}
  printf("\nThe Addition of two matrix is\n");
  for(i=0;i<row1;i++)
{
     printf("\n");
     for(j=0;j<col1;j++)
{
```

```
        printf("%d\t",c[i][j]);
   }
  }
  return 0;
}
```

P_4: Program to matrix transpose.

```
/* 2-D Transpose */
#include"stdio.h"
#include"conio.h"
void main()
{
int r,c,i,j;
int a[10][10],b[10][10];
printf("\n Enter the row size:");
scanf("%d",&row);
printf("\n Enter the column size:");
scanf("%d", &col);
for(i=0;i<row;i++)
{
for(j=0;j<col;j++)
{
printf("\n matrix [%d][%d]", i,j);
scanf("%d",&a[i][j]);
}
}
for(i=0;i<row;i++)
{
for(j=0;j<col;j++)
```

```
{
b[j][i]=a[i][j];
}
}
printf("\n Transpose of given Matrixx::");

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


P_5: To search the location of given data in the array / Linear Search [SEARCHING]

```
#include<stdio.h>
#include<conio.h>
void main()
{
int i,n,data,a[20],f=0;
printf("\n Enter the Array Capacity::");
scanf("%d", &n);
for(i=0;i<n;i++)
{
printf("\n Enter the %d Data::", i+1);
scanf("%d", &a[i]);
}
```

```
printf("\n Enter the data to be searched::");
scanf("%d", &data);
for(i=0;i<n;i++)
{
if(data==a[i])
{
printf("\n Data is found in %d location", i+1);
f=1;
}
}
if(f!=1)
{
printf("\n Data not Found::");
}
getch();
}
```


P_6: Binary Search in C:

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

int main()
{
  int i, first, last, middle, n, search, array[100];

  printf("Enter number of elements\n");
  scanf("%d",&n);

  printf("Enter %d integers\n", n);

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

  printf("Enter value to find\n");
  scanf("%d",&search);

  first = 0;
```

```
  last = n - 1;
  middle = (first+last)/2;

  while( first <= last )
  {
    if ( array[middle] < search )
       first = middle + 1;
    else if ( array[middle] >search )
    {
     last=middle+1;

    }
    else
      printf("%d found at location %d.\n", search, middle+1);
      }

  if ( first > last )
    printf("Not found! %d is not present in the list.\n", search);

  return 0;
}
```


EXPECTED ANNA UNIVERSITY PART-B QUESTIONS

1. Write short notes on: 1D arrays and 2D Arrays. [10]

2. Difference between numeric arrays and character arrays. [6]

3. Explain in detail about string functions in C. [8]

4. Describe about the declaration and initialization of arrays. [8]

AND

ALL PROGRAMS GIVEN IN THE NOTES

ALL THE BEST

UNIT IV FUNCTIONS AND POINTERS

Function – definition of function – Declaration of function – Pass by value – Pass by reference –Recursion – Pointers - Definition – Initialization – Pointers arithmetic – Pointers and arrays- Example Problems.

4. FUNCTION

- Functions are created when the same process or an algorithm to be repeated several times in various places in the program.
- Function has a self-contained block of code, that executes certain task. A function has a name, a list of arguments which it takes when called, and the block of code it executes when called.
- Functions are two types:
  - ✓ Built-in / Library function.
    ex:) printf(), scanf(), getch(), exit(), etc.
  - ✓ User defined function.

User-Defined Functions

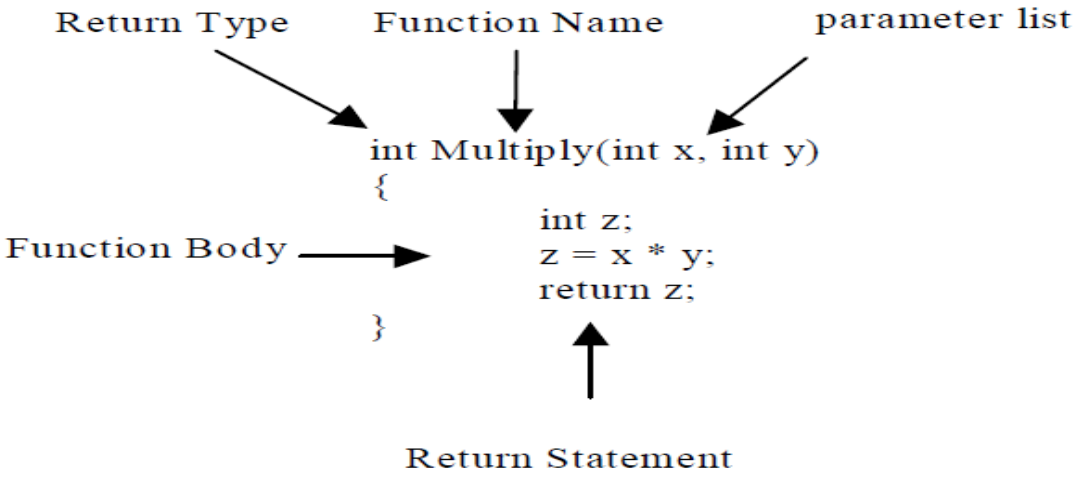
Functions defined by the users according to their requirements are called user-defined functions. These functions are used to break down a large program into a small functions.

Advantage of User defined function:

1. Reduce the source code
2. Easy to maintain and modify
3. It can be called any where in the program.

Body of user defined function:

```
return_type f_name (argument1,argument 2)
{
 local variables;
statements;
return_type;
}
```



The body of user-defined shows that an user-defined functions has following characteristics:

1. Return type
2. Function name
3. Parameter list of arguments
4. Local variables
5. Statements
6. Return value

S.no	C function aspects	syntax
1	function definition	return_type function_name (arguments list) { Body of function; }
2	function call	function_name (arguments list);
3	function declaration	return_type function_name (argument list);

Note: The function with return value must be the data type, that is return type and return value must be of same data type.

User defined function has three parts:

1. Declaration part

ret_type f_name (arguments)

2. Calling part

f_name(arguments);

3. Definition part

Function Parameters:

Parameters provide the data communication between calling function and called function.

Two types:

- ***Actual parameters:***

  These are the parameters transferred from the calling function[main function] to the called function[user defined function].

- ***Formal parameters:***

  Passing the parameters from the called functions[user defined function] to the calling functions[main function].

Note:

- Actual parameter – This is the argument which is used in function call.
- Formal parameter – This is the argument which is used in function definition.

4.1 Categories of User defined function/Function Prototypes:

- A function prototype declaration consists of the function's return type, name and arguments list.
- Always terminated with semicolon. The following are the function prototypes:

1. Function without return value and without argument.

2. Function without return value and with argument.

3. Function with return value and with argument.

4. Function with return value and without argument.

Note: Function with more than one return value will not have return type and return statement in the function definition.

Consider the following example to multiply two numbers:

```
void main( )
{
 int x,y,z;
scanf("%d%d", &x,&y);
z=x* y;
printf("The result is %d", z);
}
```

1. Function without return value and without argument

In this prototype, no data transfers takes place between the calling function and the called function. They read data values and print result in the same block.

Syntax:

```
void f_name();    //function declaration

void f_name (void)    //function definition
{
 local variables;
statements;
}
void main()
{
f_name();   //function call
}
```

The above example can be rewritten by creating function:

```
void f_mult(); //function definition
void f_mult(void )
{
 int x,y,z;
scanf("%d%d", &x,&y);
z=x* y;
printf("The result is %d", z);
}
void main()
{
f_mult();
}
```

2. Function without return value and with argument

In this prototype, data is transferred from the calling function to called function. The called function receives some data from the calling function and does not send back any values to the calling functions.

Syntax:

```
void f_name(int x, int y );   //Function declaration
void f_name (int x, int y)         //Function Definition  //Formal Parameters
{
 local variables;
statements;
}
void main()
{
//variable declaration
//input statement
f_name(c, d);   //Function call   //Actual Parameters
}
```

The above example can be rewritten by creating function

```
void f_mult(int x, int y);
void f_mult(int x, int y )
{
 int z;
z=x* y;
printf("The result is %d", z);
}
void main()
{
int c, d;
printf("Enter any two number");
scanf("%d%d", &c, &d);
f_mult(c, d);   //Function call
}
```


3. Function with return value and without argument

The calling function cannot pass any arguments to the called function but the called function may send some return value to the calling function.

Syntax:

```
int f_name();  //Function declaration
int f_name (void)          //Function Definition
{
 local variables;
statements;
return int;
}
void main()
{
//variable declaration
```

```
ret_var=f_name();   //Function call
}
```

The above example can be rewritten by creating function

```
int f_mult();
int f_mult(void )
{
 int x,y,z;
scanf("%d%d", &x,&y);
z=x* y;
printf("The result is %d", z);
return(z); }
void main()
{
int c;
c=f_mult();
getch();
}
```


4. Function with return value and with argument

In this prototype, the data is transferred between the calling function and called function. The called function receives some data from the calling function and send back a value return to the calling function.

Syntax:

```
int f_name (int x, int y);  //Function declaration
int f_name (int x, int y)        //Function definition  //Formal Parameters
{
 local variables;
statements;
return int;
```

```
}
void main()
{
//variable declaration
//Input Statement
ret_value=f_mult(a, b);                    //Function Call  //Actual Parameters
}
```

The above example can be rewritten by creating function:

```
int f_mult(int x, int y);
int f_mult(int x, int y)
{
 int z;
z=x* y;
printf("The result is %d", z);
return(z);
}
void main()
{
int a,b,c;
printf("Enter any two value:");
scanf("%d%d", &a, &b);
c=f_mult(a, b);
}
```


Note:

- If the return data type of a function is "void", then, it can't return any values to the calling function.
- If the return data type of the function is other than void such as "int, float, double etc", then, it can return values to the calling function.

return statement:

It is used to return the information from the function to the calling portion of the program.

Syntax:

return;
return();
return(constant);
return(variable);
return(exp);
return(condn_exp);

By default, all the functions return int data type.

Do you know how many values can be return from C functions?

- Always, only one value can be returned from a function.
- If you try to return more than one values from a function, only one value will be returned that appears at the right most place of the return statement.
- For example, if you use "return a,b,c" in your function, value for c only will be returned and values a, b won't be returned to the program.
- In case, if you want to return more than one values, pointers can be used to directly change the values in address instead of returning those values to the function.

Function Call:

A function can be called by specifying the function_name in the source program with parameters, if presence within the paranthesis.

Syntax:

Fn_name();
Fn_name(parameters);
Ret_value=Fn_name(parameters);

Example:

```
#include<stdio.h>
#include<conio.h>
int add(int a, int b);                          //function declaration
void main()
{
int x,y,z;
printf("\n Enter the two values:");
scanf("%d%d",&x,&y);
z=add(x,y);                                     //Function call(Actual parameters)
printf("The sum is .%d", z);
}
int add(int a, int b)                           //Function definition(Formal parameters)
{
int c;
c=a+b;
return(c);                                      //return statement
}
```


4.2 Parameter Passing Methods/Argument Passing Methods

Call by Value/Pass by Value:

When the value is passed directly to the function it is called call by value. In call by value only a copy of the variable is only passed so any changes made to the variable does not reflects in the calling function.

Example:

```
#include<stdio.h>
#include<conio.h>
swap(int,int);
void main()
{
int x,y;
printf("Enter two nos");
scanf("%d %d",&x,&y);
printf("\nBefore swapping : x=%d y=%d",x,y);
swap(x,y);
getch();
}
swap(int a,int b)
{
int t;
t=a;
a=b;
b=t;
printf("\nAfter swapping :x=%d y=%d",a,b);
}
```


System Output:

Enter two nos 12 34

Before swapping :12 34

After swapping : 34 12

Call by Reference/Pass by Reference:

When the address of the value is passed to the function it is called call by reference. In call by reference since the address of the value is passed any changes made to the value reflects in the calling function**.**

Example:

```
#include<stdio.h>
#include<conio.h>
swap(int *, int *);
void main()
{
int x,y;
printf("Enter two nos");
scanf("%d %d",&x,&y);
printf("\nBefore swapping:x=%d y=%d",x,y);
swap(&x,&y);
printf("\nAfter swapping :x=%d y=%d",x,y);
getch();
}
swap(int *a,int *b)
{
int t;
t=*a;
*a=*b;
*b=t;
}
```

System Output:

```
Enter two nos 12 34
Before swapping :12 34
After swapping : 34 12
```

Call by Value	Call by Reference
This is the usual method to call a function in which only the value of the variable is passed as an argument	In this method, the address of the variable is passed as an argument
Any alternation in the value of the argument passed does not affect the function.	Any alternation in the value of the argument passed affect the function.
Memory location occupied by formal and actual arguments is different	Memory location occupied by formal and actual arguments is same and there is a saving of memory location
Since a new location is created, this method is slow	Since the existing memory location is used through its address, this method is fast
There is no possibility of wrong data manipulation since the arguments are directly used in an application	There is a possibility of wrong data manipulation since the addresses are used in an expression. A good skill of programming is required here

Library Functions:

C language provides built-in-functions called library function compiler itself evaluates these functions.

List of Functions

- ✓ Sqrt(x)→ $(x)^{0.5}$
- ✓ Log(x)
- ✓ Exp(x)
- ✓ Pow(x,y)
- ✓ Sin(x)
- ✓ Cos(x)
- ✓ Rand(x)-> generating a positive random integer.

4.3 Recursion in C:

Recursion is calling function by itself again and again until some specified condition has been satisfied.

Syntax:

```
int f_name (int x)
{
 local variables;
f_name(y); // this is recursion
statements;
}
```

Example_1: Factorial using Recursion

```
#include<stdio.h>
#include<conio.h>
int factorial(int n);
void main()
{
int res,x;
printf("\n Enter the value:");
scanf("%d", &x);
res=factorial(x);
printf("The factorial of %d is ..%d", res);
}
int factorial(int n)
{
 int fact;
if (n==1)
return(1);
else
fact = n*factorial(n-1);
return(fact);    }
```

Example_2: Fibonacci using Recursion

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

int Fibonacci(int n)
{
if ( n == 0 )
return 0;
else if ( n == 1 )
return 1;
else
return ( Fibonacci(n-1) + Fibonacci(n-2) );
}
```

Example_3: Sum of n Natural Numbers

```
#include<stdio.h>
#include<conio.h>
 int add(int n);
void main()
{
 int m, x;
 printf("Enter an positive integer: ");
 scanf("%d",&m);
x=add(m);
printf("Sum = %d", x);
 getch();
}
 int add(int n)
{
if(n!=0)
 return n+add(n-1); /* recursive call */
}
```


4.4 POINTERS

Definition:

- C Pointer is a variable that stores/points the address of the another variable.
- C Pointer is used to allocate memory dynamically i.e. at run time.
- The variable might be any of the data type such as int, float, char, double, short etc.
- Syntax : data_type *var_name;
  Example : int *p; char *p;
  Where, * is used to denote that “p” is pointer variable and not a normal variable.

Key points to remember about pointers in C:

- Normal variable stores the value whereas pointer variable stores the address of the variable.
- The content of the C pointer always be a whole number i.e. address.
- Always C pointer is initialized to null, i.e. int *p = null.
- The value of null pointer is 0.
- & symbol is used to get the address of the variable.
- * symbol is used to get the value of the variable that the pointer is pointing to.
- If pointer is assigned to NULL, it means it is pointing to nothing.
- The size of any pointer is 2 byte (for 16 bit compiler).

- No two pointer variables should have the same name.
- But a pointer variable and a non-pointer variable can have the same name.

4.4.1 Pointer –Initialization:

Assigning value to pointer:

It is not necessary to assign value to pointer. Only zero (0) and NULL can be assigned to a pointer no other number can be assigned to a pointer. Consider the following examples;

int *p=0;
int *p=NULL; The above two assignments are valid.
int *p=1000; This statement is invalid.

Assigning variable to a pointer:

int x; *p;
p = &x;

This is nothing but a pointer variable p is assigned the address of the variable x. The address of the variables will be different every time the program is executed.

Reading value through pointer:

int x=123; *p;

p = &x;

Here the pointer variable p is assigned the address of variable x.

printf("%d", *p); will display value of x 123. This is reading value through pointer

printf("%d", p); will display the address of the variable x.

printf("%d", &p); will display the address of the pointer variable p.

printf("%d",x); will display the value of x 123.

printf("%d", &x); will display the address of the variable x.

Note: It is always a good practice to assign pointer to a variable rather than 0 or NULL.

Pointer Assignments:

We can use a pointer on the right-hand side of an assignment to assign its value to another variable.

Example:

```
int main()
{
int var=50;
int *p1, *p2;
p1=&var;
p2=p1;
}
```

Chain of pointers/Pointer to Pointer:

A pointer can point to the address of another pointer. Consider the following example.

```
int x=456, *p1, **p2;          //[pointer-to-pointer];
p1 = &x;
p2 = &p1;
```

printf("%d", *p1); will display value of x 456.

printf("%d", *p2); will also display value of x 456. This is because p2 point p1, and p1 points x. Therefore p2 reads the value of x through pointer p1. Since one pointer is points towards another pointer it is called chain pointer. Chain pointer must be declared with ** as in **p2.

Manipulation of Pointers

We can manipulate a pointer with the indirection operator '*', which is known as dereference operator. With this operator, we can indirectly access the data variable content.

Syntax:

```
*ptr_var;
```


Example:

```
#include<stdio.h>
void main()
{
int a=10, *ptr;
ptr=&a;
printf("\n The value of a is ",a);
*ptr=(*ptr)/2;
printf("The value of a is.",(*ptr));
}
```

Output:

The value of a is: 10

The value of a is: 5

4.4.2 Pointer Expression & Pointer Arithmetic

C allows pointer to perform the following arithmetic operations:

A pointer can be incremented / decremented.

Any integer can be added to or subtracted from the pointer.

A pointer can be incremented / decremented.

In *16 bit machine*, size of all types[data type] of pointer always 2 bytes.

Eg: int a;

int *p;

p++;

Each time that a pointer p is incremented, the pointer p will points to the memory location of the next element of its base type. Each time that a pointer p is decremented, the pointer p will points to the memory location of the previous element of its base type.


```
int a,*p1, *p2, *p3;
p1=&a;
p2=p1++;
p3=++p1;

printf("Address of p where it points to %u", p1);     1000
printf("After incrementing Address of p where it points to %u", p1);     1002
printf("After assigning and incrementing p  %u", p2);     1000
printf("After incrementing and assigning p  %u", p3);     1002
```

In *32 bit machine*, size of all types of pointer is always 4 bytes.

The pointer variable p refers to the base address of the variable a. We can increment the pointer variable,

p++ or ++p

This statement moves the pointer to the next memory address. let p be an integer pointer with a current value of 2,000 (that is, it contains the address 2,000). Assuming 32-bit integers, after the expression

```
p++;
```

the contents of p will be 2,004, not 2,001! Each time p is incremented, it will point to the next integer. The same is true of decrements. For example,

```
p--;
```

will cause p to have the value 1,996, assuming that it previously was 2,000. Here is why: Each time that a pointer is incremented, it will point to the memory location of the next element of its base type. Each time it is decremented, it will point to the location of the previous element of its base type.

Any integer can be added to or subtracted from the pointer.

Like other variables pointer variables can be used in expressions. For example if p1 and p2 are properly declared and initialized pointers, then the following statements are valid.

```
y=*p1**p2;
sum=sum+*p1;
z= 5* - *p2/p1;
*p2= *p2 + 10;
```

C allows us to add integers to or subtract integers from pointers as well as to subtract one pointer from the other. We can also use short hand operators with the pointers p1+=; sum+=*p2; etc., we can also compare pointers by using relational operators the expressions such as p1 >p2 , p1==p2 and p1!=p2 are allowed.

/*Program to illustrate the pointer expression and pointer arithmetic*/

```
#include< stdio.h >
#include<conio.h>
void main()
{
int ptr1,ptr2;
int a,b,x,y,z;
a=30;b=6;
ptr1=&a;
ptr2=&b;
x=*ptr1+ *ptr2 –6;
y=6*- *ptr1/ *ptr2 +30;
printf(“\nAddress of a + %u”,ptr1);
printf(“\nAddress of b   %u”, ptr2);
printf(“\na=%d, b=%d”, a, b);
printf(“\nx=%d,y=%d”, x, y);
ptr1=ptr1 + 70;
ptr2= ptr2;
printf(“\na=%d, b=%d”, a, b);
}
```


/* Sum of two integers using pointers*/

```
#include <stdio.h>
 int main()
{
   int first, second, *p, *q, sum;
    printf("Enter two integers to add\n");
    scanf("%d%d", &first, &second);
    p = &first;
   q = &second;
```

```
  sum = *p + *q;
  printf("Sum of entered numbers = %d\n",sum);
 return 0;
}
```

4.4.3 Pointers and Arrays

Array name is a constant pointer that points to the base address of the array[i.e the first element of the array]. Elements of the array are stored in contiguous memory locations. They can be efficiently accessed by using pointers.

Pointer variable can be assigned to an array. The address of each element is increased by one factor depending upon the type of data type. The factor depends on the type of pointer variable defined. If it is integer the factor is increased by 2. Consider the following example:


```
int x[5]={11,22,33,44,55}, *p;
p = x;          //p=&x;    // p = &x[0];
```

Remember, earlier the pointer variable is assigned with address (&) operator. When working with array the pointer variable can be assigned as above or as shown below:

Therefore the address operator is required only when assigning the array with element. Assume the address on x[0] is 1000 then the address of other elements will be as follows

x[1] = 1002

x[2] = 1004

x[3] = 1006

x[4] = 1008

The address of each element increase by factor of 2. Since the size of the integer is 2 bytes the memory address is increased by 2 bytes, therefore if it is float it will be increase 4 bytes, and for double by 8 bytes. This uniform increase is called scale factor.

p = &x[0];

Now the value of pointer variable p is 1000 which is the address of array element x[0]. To find the address of the array element x[1] just write the following statement.

p = p + 1;

Now the value of the pointer variable p is 1002 not 1001 because since p is pointer variable the increment of will increase to the scale factor of the variable, since it is integer it increases by 2.

The p = p + 1; can be written using increment or decrement operator ++p; The values in the array element can be read using increment or decrement operator in the pointer variable using scale factor.

Consider the above example.

printf("%d", *(p+0)); will display value of array element x[0] which is 11.

printf("%d", *(p+1)); will display value of array element x[1] which is 22.

printf("%d", *(p+2)); will display value of array element x[2] which is 33.

printf("%d", *(p+3)); will display value of array element x[3] which is 44.

printf("%d", *(p+4)); will display value of array element x[4] which is 55.

/*Displaying the values and address of the elements in the array*/

```
#include<stdio.h>
void main()
{
int a[6]={10, 20, 30, 40, 50, 60};
int *p;
int i;
p=a;
```

```
for(i=0;i<6;i++)
{
printf(“%d”, *p);  //value of elements of array
printf(“%u”,p);  //Address of array
}
getch();
}
```

/* Sum of elements in the Array*/

```
#include<stdio.h>
#include<conio.h>
void main()
{
 int a[10];
 int i,sum=0;
 int *ptr;

 printf("Enter 10 elements:n");

 for(i=0;i<10;i++)
    scanf("%d",&a[i]);
 ptr = a;          /* a=&a[0] */
 for(i=0;i<10;i++)
    {
    sum = sum + *ptr;    //*p=content pointed by 'ptr'
    ptr++;
    }
 printf("The sum of array elements is %d",sum);
}
```

/*Sort the elements of array using pointers*/

```
#include<stdio.h>
int main(){
   int  i,j, temp1,temp2;
   int arr[8]={5,3,0,2,12,1,33,2};
   int *ptr;
   for(i=0;i<7;i++){
      for(j=0;j<7-i;j++){
         if(*(arr+j)>*(arr+j+1)){
            ptr=arr+j;
            temp1=*ptr++;
            temp2=*ptr;
            *ptr--=temp1;
            *ptr=temp2;
         }}}
for(i=0;i<8;i++){
      printf(" %d",arr[i]); } }
```

4.4.4 Pointers and Multi-dimensional Arrays

The array name itself points to the base address of the array.

Example:

```
int a[2][3];

int *p[2];

p=a;   //p points to a[0][0]
```

/*Displaying the values in the 2-d array*/

```
#include<stdio.h>
void main()
{
int a[2][2]={{10, 20},{30, 40}};
int *p[2];
int i,j;
p=a;
for(i=0;i<2;i++)
{
for(j=0;j<2;j++)
{
printf("%d", *(*(p+i)+j));  //value of elements of array
}
}
getch();
}
```

4.5 Dynamic Memory Allocation

The process of allocating memory during program execution is called dynamic memory allocation.

Dynamic memory allocation functions

S. No.	***Function***	***Syntax***	***Use***
1	malloc()	ptr=(cast-type*)malloc(byte-size)	Allocates requested size of bytes and returns a pointer first byte of allocated space.
2	calloc()	ptr=(cast-type*)calloc(n,element-size);	Allocates space for an array elements, initializes to zero and then returns a pointer to memory.
3	free()	free(ptr);	dellocate the previously allocated space.
4	realloc()	ptr=realloc(ptr,newsize);	Change the size of previously allocated space.

EXPECTED ANNA UNIVERSITY PART-B QUESTION

1. **Explain in detail about the function prototypes.** **[8]**
2. **Explain in detail about parameter passing methods.** **[8]**
3. **Describe the pass by value concept.** **[8]**
4. **Describe the pass by reference concept.** **[8]**
5. **Explain the concept of recursive functions in C.** **[8]**
6. **Write short notes on: pointer arithmetic.** **[6]**
7. **Explain in detail about pointers and array in C.** **[8]**

And all the Program'z given in the notes.....

UNIT V STRUCTURES AND UNIONS

Introduction – need for structure data type – structure definition – Structure declaration – Structure within a structure - Union - Programs using structures and Unions – Storage classes, Pre-processor directives.

1

Array	Structure
Single name that contains a collection of data-items of same data type.	Single name that contains a collection of data-items of different data types.
Individual entries in a array are called elements.	Individual entries in a structure are called members.
No Keyword	Keyword: struct
Members of an array are stored in sequence of memory locations.	Members of a structure are not stored in sequence of memory locations.

5.1 STRUCTURES

Structure is a collection of variables under the single name, which can be of different data type. In simple words, Structure is a convenient way of grouping several pieces of related information together.

A structure is a derived data type, The scope of the name of a structure member is limited to the structure itself and also to any variable declared to be of the structure's type.

Variables which are declared inside the structure are called "**members of structure**".

Syntax: In general terms, the composition of a structure may be defined as:

```
struct <tag>
{
 member 1;
member 2;
----------
----------
member m;
};
```

UNIT V STRUCTURES AND UNIONS Introduction – need for structure data type – structure definition – Structure declaration – Structure within a structure - Union - Programs using structures and Unions – Storage classes, Pre-processor directives.

Array Structure

Single name that contains a collection of data-

items of same data type.

Single name that contains a collection of data-items of

different data types.

Individual entries in a array are called elements. Individual entries in a structure are called members.

No Keyword Keyword: struct

Members of an array are stored in sequence of

memory locations.

Members of a structure are not stored in sequence of

memory locations.

5.1 STRUCTURES

Structure is a collection of variables under the single name, which can be of different

data type. In simple words, Structure is a convenient way of grouping several pieces of related

information together.

A structure is a derived data type, The scope of the name of a structure member is limited

to the structure itself and also to any variable declared to be of the structure's type.

Variables which are declared inside the structure are called “members of structure”.

Syntax: In general terms, the composition of a structure may be defined as:

struct <tag>

{

member 1;

member 2;

---

member m;

};

1

where, struct is a keyword, <tag> is a structure name.

For example:

```
struct student
{
char name [80];
int roll_no;
float marks;
};
```

Now we need an interface to access the members declared inside the structure, it is called *structure variable*. we can declare the structure variable in two ways:

i) within the structure definition itself.

ii) within the main function.

i]within the structure definition itself.

```
struct tag
{
member 1;
member 2;
- ——
- —-
- member m;
} variable 1, variable 2 --------- variable n;  //Structure variables
```

Eg:

```
struct student
{
char name [80];
int roll_no;
```

GE 6151 COMPUTER PROGRAMMING UNIT_5

where, struct is a keyword, <tag> is a structure name.

For example:

```
struct student
{
char name [80];
int roll_no;
float marks;
};
```

Now we need an interface to access the members declared inside the structure, it is called structure variable. we can declare the structure variable in two ways:

i) within the structure definition itself.

ii) within the main function.

i]within the structure definition itself.

```
struct tag
{
member 1;
member 2;
- ——
- —-
- member m;
} variable 1, variable 2 --------- variable n; //Structure variables
```

Eg:

```
struct student
{
char name [80];
int roll_no;
```

```
float marks;
}s1;
```

ii]within the main function.

```
struct tag
{
member 1;
member 2;
- ——
- —-
- member m;
};
void main()
{
struct tag var1, var2,…,var_n;  //structure variable
}
```


Eg:

```
struct student
{
char name [80];
int roll_no;
float marks;
};

void main()
{
struct student s1, s2;  //declaration of structure variable.
};
```

float marks;

}s1;

ii]within the main function.

struct tag

{

member 1;

member 2;

- ——

- —-

- member m;

};

void main()

{

struct tag var1, var2,...,var_n; //structure variable

}

Eg:

struct student

{

char name [80];

int roll_no;

float marks;

};

void main()

{

struct student s1, s2; //declaration of structure variable.

};

Note: Structure variable can be any of three types: normal variable, array_variable, pointer_variable.

Initialization of Structure members:

A structure variable, like an array can be initialized in two ways:

I.
```
struct student
{
char name [80];
int roll_no;
float marks ;
} s1={"Saravana",34,469};
```

II.
```
struct student
{
char name [80];
int roll_no;
float marks ;
} s1;
struct student s1= {"Saravana",34,469};
```
[OR]
```
s1.name={"Saravana"};
s1.roll_no=34;
s1.marks=500;
```

It is also possible to define an array of structure, that is an array in which each element is a structure. The procedure is shown in the following example:

```
struct student
{
char name [80];
int roll_no ;
float marks ;
} st [100];
```

GE 6151 COMPUTER PROGRAMMING UNIT_5

Note: Structure variable can be any of three types: normal variable, array_variable, pointer_variable.

Initialization of Structure members:

A structure variable, like an array can be initialized in two ways:

I. struct student

{

char name [80];

int roll_no;

float marks ;

} s1={“Saravana”,34,469};

II. struct student

{

char name [80];

int roll_no;

float marks ;

} s1;

struct student s1= {“Saravana”,34,469};

[OR]

s1.name={“Saravana”};

s1.roll_no=34;

s1.marks=500;

It is also possible to define an array of structure, that is an array in which each element is a structure. The procedure is shown in the following example:

struct student

{

char name [80];

int roll_no ;

float marks ;

} st [100];

4

In this declaration st is a 100- element array of structures. It means each element of st represents an individual student record.

Accessing the Structure Members

The members of a structure are usually processed individually, as separate entities. Therefore, we must be able to access the individual structure members. A structure member can be accessed by writing:

structure_variable.member_name *//[normal variable]*

Eg:

```
struct student
{
char name [80];
int roll_no ;
float marks ;
}st;
```


e.g. if we want to get the detail of a member of a structure then we can write as scanf("%s",st.name); or scanf("%d", &st.roll_no) and so on.

if we want to print the detail of a member of a structure then we can write as printf("%s",st.name); or printf("%d", st.roll_no) and so on.

The use of the period operator can be extended to arrays of structure by writing:

array [expression].member_name *//[array variable]*

Eg:

```
struct student
{
char name [80];
int roll_no ;
```

In this declaration st is a 100- element array of structures. It means each element of st represents an individual student record.

Accessing the Structure Members

The members of a structure are usually processed individually, as separate entities.

Therefore, we must be able to access the individual structure members. A structure member can

be accessed by writing:

structure_variable.member_name //[normal variable]

Eg:

```
struct student
{
char name [80];
int roll_no ;
float marks ;
}st;
```

e.g. if we want to get the detail of a member of a structure then we can write as

scanf("%s",st.name); or scanf("%d", &st.roll_no) and so on.

if we want to print the detail of a member of a structure then we can write as

printf("%s",st.name); or printf("%d", st.roll_no) and so on.

The use of the period operator can be extended to arrays of structure by writing:

array [expression].member_name //[array variable]

Eg:

```
struct student
{
char name [80];
int roll_no ;
```

float marks ;
}st[10];

e.g. if we want to get the detail of a member of a structure then we can write as scanf("%s",st[i].name); or scanf("%d", &st[i].roll_no) and so on.

if we want to print the detail of a member of a structure then we can write as printf("%s",st[i].name); or printf("%d", st[i].roll_no) and so on.

The use of the period operator can be extended to pointer of structure by writing:
array [expression]->member_name *//[array variable]*

Eg:

```
struct student
{
char name [80];
int roll_no ;
float marks ;
}*st;

void main()
{
Struct student s;
st=&s;
```

e.g. if we want to get the detail of a member of a structure then we can write as scanf("%s",st->name); or scanf("%d", &st->roll_no) and so on.

if we want to print the detail of a member of a structure then we can write as printf("%s",st->name); or printf("%d", st->roll_no) and so on.

float marks ;

}st[10];

e.g. if we want to get the detail of a member of a structure then we can write as scanf(“%s”,st[i].name); or scanf(“%d”, &st[i].roll_no) and so on.

if we want to print the detail of a member of a structure then we can write as printf(“%s”,st[i].name); or printf(“%d”, st[i].roll_no) and so on.

The use of the period operator can be extended to pointer of structure by writing:

array [expression]->member_name //[array variable]

Eg:

```
struct student
{
char name [80];
int roll_no ;
float marks ;
}*st;
void main()
{
Struct student s;
st=&s;
```

e.g. if we want to get the detail of a member of a structure then we can write as scanf(“%s”,st->name); or scanf(“%d”, &st->roll_no) and so on.

if we want to print the detail of a member of a structure then we can write as printf(“%s”,st->name); or printf(“%d”, st->roll_no) and so on.

It is also possible to pass entire structure to and from functions though the way this is done varies from one version of 'C' to another.

PROGRAM'S USING STRUCTURE

Example 1://To print the student details// normal structure variable

```
#include<stdio.h>
struct student
{
char name[30];
int reg_no[15];
char branch[30];
};
void main()
{
struct student s1;
printf("\n Enter the student name::");
scanf("%s",s1.name);
printf("\n Enter the student register number::");
scanf("%s",&s1.reg_no);
printf("\n Enter the student branch::");
scanf("%s",s1.branch);
printf("\n Student Name::",s1.name);
printf("\n Student Branch::",s1.branch);
printf("\n Student reg_no::",s1.reg_no);
getch();
}
```

GE 6151 COMPUTER PROGRAMMING UNIT_5

It is also possible to pass entire structure to and from functions though the way this is done varies from one version of 'C' to another.

PROGRAM'S USING STRUCTURE

Example 1://To print the student details// normal structure variable

```
#include<stdio.h>
struct student
{
char name[30];
int reg_no[15];
char branch[30];
};
void main()
{
struct student s1;
printf("\n Enter the student name::");
scanf("%s",s1.name);
printf("\n Enter the student register number::");
scanf("%s",&s1.reg_no);
printf("\n Enter the student branch::");
scanf("%s",s1.branch);
printf("\n Student Name::",s1.name);
printf("\n Student Branch::",s1.branch);
printf("\n Student reg_no::",s1.reg_no);
getch();
}
```

Example 2://To print the student details// pointer structure variable

```
#include <stdio.h>
struct name{
 int a;
 float b;
};
 int main()
{
struct name *ptr,p;
 ptr=&p;                    /* Referencing pointer to memory address of p */
printf("Enter integer: ");
 scanf("%d",&(*ptr).a);
 printf("Enter number: ");
scanf("%f",&(*ptr).b);
 printf("Displaying: ");
 printf("%d%f",(*ptr).a,(*ptr).b);
 return 0;
}
```


Example 3://To print the Mark sheet for a student//

```
#include<stdio.h>
struct student
{
char name[30];
int reg_no[15];
char branch[30];
int m1,m2,m3,total;
float avg;
};
```

GE 6151 COMPUTER PROGRAMMING UNIT_5

Example 2://To print the student details// pointer structure variable

```
#include <stdio.h>
struct name{
int a;
float b;
};
int main()
{
struct name *ptr,p;
ptr=&p; /* Referencing pointer to memory address of p */
printf("Enter integer: ");
scanf("%d",&(*ptr).a);
printf("Enter number: ");
scanf("%f",&(*ptr).b);
printf("Displaying: ");
printf("%d%f",(*ptr).a,(*ptr).b);
return 0;
}
```

Example 3://To print the Mark sheet for a student//

```
#include<stdio.h>
struct student
{
char name[30];
int reg_no[15];
char branch[30];
int m1,m2,m3,total;
float avg;
};
```

8

```
void main()
{
int total;
float avg;
struct student s1;
printf("\n Enter the student name::");
scanf("%s",s1.name);
printf("\n Enter the student register number::");
scanf("%s",&s1.reg_no);
printf("\n Enter the student branch::");
scanf("%s",s1.branch);
printf("\n Enter the 3 subjects Marks:");
scanf("%d%d%d", &s1.m1,&s1.m2,&s1.m3);
s1.total=s1.m1+s1.m2+s1.m3;
printf("\n Ur Total mark is.%d", s1.total);
s1.avg=a1.total/3;
printf("\n Ur Average mark is.%f", s1.avg);
getch();
}
```


Example_4: /* To Print Students Mark Sheet's using Structures*/

```
#include"stdio.h"
#include"conio.h"
struct student
{
char name[25],grade;
int reg_no[15];
int s1,s2,s3,s4,s5,total;
float avg;
}sa[20];
```

GE 6151 COMPUTER PROGRAMMING UNIT_5

```
void main()
{
int total;
float avg;
struct student s1;
printf("\n Enter the student name::");
scanf("%s",s1.name);
printf("\n Enter the student register number::");
scanf("%s",&s1.reg_no);
printf("\n Enter the student branch::");
scanf("%s",s1.branch);
printf("\n Enter the 3 subjects Marks:");
scanf("%d%d%d", &s1.m1,&s1.m2,&s1.m3);
s1.total=s1.m1+s1.m2+s1.m3;
printf("\n Ur Total mark is.%d", s1.total);
s1.avg=a1.total/3;
printf("\n Ur Average mark is.%f", s1.avg);
getch();
}
```

Example_4: /* To Print Students Mark Sheet's using Structures*/

```
#include"stdio.h"
#include"conio.h"
struct student
{
char name[25],grade;
int reg_no[15];
int s1,s2,s3,s4,s5,total;
float avg;
```

}sa[20];

9

```
void main()
{
int i,n,total;
float avg;
printf("\n Enter the count of Students need marksheet::");
scanf("%d",&n);
printf("\n Enter the name of students:,reg_no, 5 sub marks::");
for(i=0;i<n;i++)
{
printf("\n Enter the name of students,reg_no, 5 sub marks::%d",i+1);
scanf("%s%d%d%d%d%d%d", &sa[i].name, &sa[i].reg_no, &sa[i].s1, &sa[i].s2, &sa[i].s3,
&sa[i].s4, &sa[i].s5);
sa[i].total=sa[i].s1+sa[i].s2+sa[i].s3+sa[i].s4+sa[i].s5;
sa[i].avg=sa[i].total/5;
if((sa[i].s1<50)||(sa[i].s2<50)||(sa[i].s3<50)||(sa[i].s4<50)||(sa[i].s5<50))
{
sa[i].grade='U';
printf("Ur grade is %c", sa[i].grade);
}
else
{
if(sa[i].avg==100)
{
sa[i].grade='S';
printf("Ur Grade is %c", sa[i].grade);
}
if((sa[i].avg>90)&&(sa[i].avg<=99))
{
sa[i].grade='A';
printf("Ur Grade is %c", sa[i].grade);
}
```

GE 6151 COMPUTER PROGRAMMING UNIT_5

```
void main()
{
int i,n,total;
float avg;
printf("\n Enter the count of Students need marksheet::");
scanf("%d",&n);
printf("\n Enter the name of students:,reg_no, 5 sub marks::");
for(i=0;i<n;i++)
{
printf("\n Enter the name of students,reg_no, 5 sub marks::%d",i+1);
scanf("%s%d%d%d%d%d", &sa[i].name, &sa[i].reg_no, &sa[i].s1, &sa[i].s2, &sa[i].s3,
&sa[i].s4, &sa[i].s5);
sa[i].total=sa[i].s1+sa[i].s2+sa[i].s3+sa[i].s4+sa[i].s5;
sa[i].avg=sa[i].total/5;
if((sa[i].s1<50)||(sa[i].s2<50)||(sa[i].s3<50)||(sa[i].s4<50)||(sa[i].s5<50))
{
sa[i].grade='U';
printf("Ur grade is %c", sa[i].grade);
}
else
{
if(sa[i].avg==100)
{
sa[i].grade='S';
printf("Ur Grade is %c", sa[i].grade);
}
if((sa[i].avg>90)&&(sa[i].avg<=99))
{
```

```
sa[i].grade='A';
printf("Ur Grade is %c", sa[i].grade);
}
```

10