NADAR SARASWATHI COLLEGE OF ENGINEERING & TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

COURSE

MATERIAL

GE6151 COMPUTER PROGRAMMING

SYLLABUS

GE6151 COMPUTER PROGRAMMING

UNIT I INTRODUCTION **8**

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.

UNIT II C PROGRAMMING BASICS **10**

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 andBranching – Looping statements – solving simple scientific and statistical problems.

UNIT III ARRAYS AND STRINGS **9**

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

UNIT IV FUNCTIONS AND POINTERS **9**

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

UNIT V STRUCTURES AND UNIONS **9**

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.

TOTAL: 45 PERIODS

TEXTBOOKS: 1. Anita Goel and Ajay Mittal, "Computer Fundamentals and Programming in C", Dorling Kindersley(India) Pvt. Ltd., Pearson Education in South Asia, 2011.

2. Pradip Dey, Manas Ghosh, "Fundamentals of Computing and Programming in C", First Edition,Oxford University Press, 2009

3. Yashavant P. Kanetkar. " Let Us C", BPB Publications, 2011.

REFERENCES:

1. Byron S Gottfried, "Programming with C", Schaum's Outlines, Second Edition, Tata McGraw-Hill,2006.

2. Dromey R.G., "How to Solve it by Computer", Pearson Education, Fourth Reprint, 2007.

3. Kernighan,B.W and Ritchie,D.M, "The C Programming language", Second Edition, PearsonEducation, 2006.

UNIT I 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.

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.

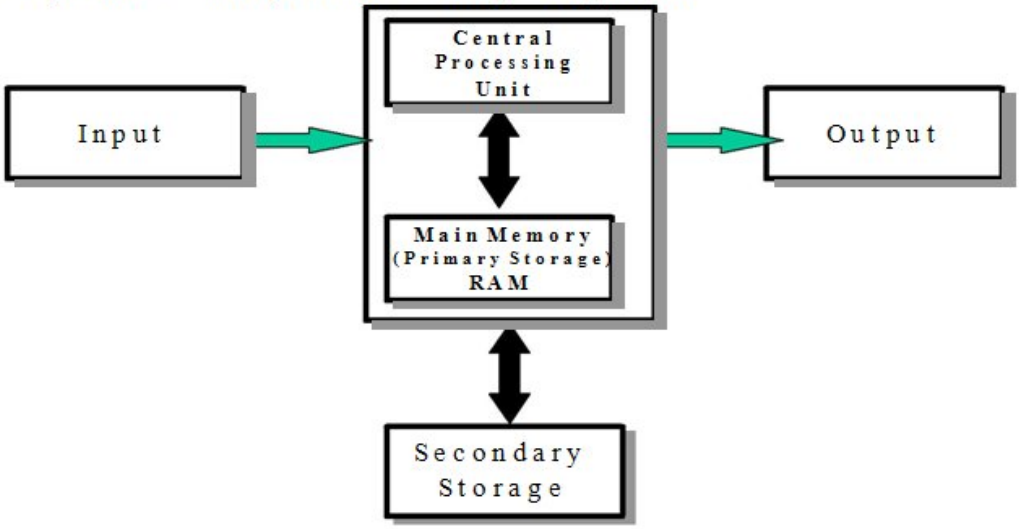
Graphic User Interface (GUI)

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. Memory4. Storage Devices



Introduction to CPU

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

CPU Operation

The fundamental operation of most CPUs

- To execute a sequence of stored instructions called a program.

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

- **Decode :**
  - The instruction is broken up into parts that have significance to other portions of the CPU.
  - The way in which the numerical instruction value is interpreted is defined by the CPU's instruction set architecture (ISA).
  - Opcode, indicates which operation to perform.
  - The remaining parts of the number usually provide information required for that instruction, such as operands for an addition operation.
  - 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 :**
  - During this step, various portions of the CPU are connected so they can perform the desired operation.
  - 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.
  - The inputs provide the numbers to be added, and the outputs will contain the final sum.
  - 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 :**
  - Simply "writes back" the results of the execute step to some form of memory.
  - Very often the results are written to some internal CPU register for quick access by subsequent instructions.
  - 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. Optical Mark Reader (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**, or **internal memory**, is computer memory that is accessible to the central processing unit of a computer without the use of computer's input/output channels
- Primary storage, also known as main storage or memory, is the main area in a computer in which data is stored for quick access by the computer's processor.

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

Printers

Printers are hardware devices that allow you to create a hard copy of a file. Today a printer is a necessary requirement for any home user and business. Allowing individuals to save their work in the format of paper instead of electronically.

Types of Printers

- **Impact printers**
  - In case of Impact printer an inked ribbon exists between the print head and paper ,the head striking the ribbon prints the character.
- **Non Impact Printers**
  - Non Impact printers use techniques other than the mechanical method of head striking the ribbon

Impact printers

Impact printers are basically divided into 2 types

- Serial/Character printers
  - Dot matrix printers
- Daisy wheel printers
  - Line Printers

Non-Impact Printers

Non Impact Printers are divided into 3 categories

- Thermal printers
- Ink jet printers
- Laser printers

Classification

Printers are classified by the following characteristics:

Quality of type: The output produced by printers is said to be either *letter quality* (as good as a typewriter), *near letter quality*, or *draft quality*. Only daisy-wheel, ink-jet, and laser printers produce letter-quality type. Some dot-matrix printers claim letter-quality print, but if you look closely, you can see the difference.

Speed: Measured in characters per second (cps) or pages per minute (ppm), the speed of printers varies widely. Daisy-wheel printers tend to be the slowest, printing about 30 cps. Line printers are fastest (up to 3,000 lines per minute). Dot-matrix printers can print up to 500 cps, and laser printers range from about 4 to 20 text pages per minute.

Impact or non-impact: Impact printers include all printers that work by striking an ink ribbon. Daisy-wheel, dot-matrix, and line printers are impact printers. Non-impact printers include laser printers and ink-jet printers. The important difference between impact and non-impact printers is that impact printers are much noisier.

Graphics: Some printers (daisy-wheel and line printers) can print only text. Other printers can print both text and graphics.

Fonts: Some printers, notably dot-matrix printers, are limited to one or a few fonts. In contrast, laser and ink-jet printers are capable of printing an almost unlimited variety of fonts. Daisy-wheel printers can also print different fonts, but you need to change the daisy wheel, making it difficult to mix fonts in the same document.

Dot Matrix Printers

A dot matrix printer or impact matrix printer refers to a type of computer printer with a print head that runs back and forth on the page and prints by impact, striking an ink-soaked cloth ribbon against the paper, much like a typewriter. Unlike a typewriter or daisy wheel printer, letters are drawn out of a dot matrix, and thus, varied fonts and arbitrary graphics can be produced. Because the printing involves mechanical pressure, these printers can create carbon copies and carbonless copies. The standard of print obtained is poor. These printers are cheap to run and relatively fast.

The moving portion of the printer is called the print head, and prints one line of text at a time. Most dot matrix printers have a single vertical line of dot-making equipment on their print heads; others have a few interleaved rows in order to improve dot density. The print head consists of 9 or 24 pins each can move freely within the tube; more the number of pins better are the quality of output. Dot Matrix Printer Characters are formed from a matrix of dots.

The speed is usually 30 - 550 characters per second (cps). These types of printers can print graphs also. They can only print text and graphics, with limited color performance. Impact printers have one of the lowest printing costs per page. These machines can be highly durable, but eventually wear out. Ink invades the guide plate of the print head, causing grit to adhere to it; this grit slowly causes the channels in the guide plate to wear from circles into ovals or slots, providing less and less accurate guidance to the printing wires. After about a million characters, even with tungsten blocks and titanium pawls, the printing becomes too unclear to read.

Daisy Wheel Printer

A daisy wheel printer is a type of computer printer that produces high-quality type, and is often referred to as a letter-quality printer (this in contrast to high-quality dot-matrix printers, capable of near-letter-quality, or NLQ, output). There were also, and still are daisy wheel typewriters, based on the same principle. The DWP is slower the speed range is in 30 to 80 CPS.

The system used a small wheel with each letter printed on it in raised metal or plastic. The printer turns the wheel to line up the proper letter under a single pawl which then strikes the back of the letter and drives it into the paper. In many respects the daisy wheel is similar to a standard typewriter in the way it forms its letters on the page, differing only in the details of the mechanism (daisy wheel vs typebars or the type ball used on IBMs electric typewriters).

Daisy wheel printers were fairly common in the 1980s, but were always less popular than dot matrix printers (ballistic wire printers) due to the latter's ability to print graphics and different fonts. With the introduction of high quality laser printers and inkjet printers in the later 1980s daisy wheel systems quickly disappeared but for the small remaining typewriter market.

Line Printer

The **line printer** is a form of high speed impact printer in which a line of type is printed at a time.
The wheels spin at high speed and paper and an inked ribbon are stepped (moved) past the print position. As the desired character for each column passes the print position, a hammer strikes the paper and ribbon causing the desired character to be recorded on the continuous paper. The speed is 300 to 2500 lines per minute (LPM). This technology is still in use in a number of applications. It is usually both faster and less expensive (in total ownership) than laser printers. In printing box labels, medium volume accounting and other large business applications, line printers remain in use

Line printers, as the name implies, print an entire line of text at a time. Two principle designs existed. In *drum printers*, a drum carries the entire character set of the printer repeated in each column that is to be printed. In *chain printers* (also known as *train printers*), the character set is arranged multiple times around a chain that travels horizontally pas the print line. In either case, to print a line, precisely timed hammers strike against the back of the paper at the exact moment that the correct character to be printed is passing in front of the paper. The paper presses forward against a ribbon which then presses against the character form and the impression of the character form is printed onto the paper.

These printers were the fastest of all impact printers and were used for bulk printing in large computer centers. They were virtually never used with personal computers and have now been partly replaced by high-speed laser printers.

Thermal Printers

Direct thermal printers create an image by selectively heating coated paper when the paper passes over the thermal print head. The coating turns black in the areas where it is heated, creating the image. More recently, two-color direct thermal printers have been produced, which allow printing of both red (or another color) and black by heating to different temperatures.

Thermal Printer Characters are formed by heated elements being placed in contact with special heat sensitive paper forming darkened dots when the elements reach a critical temperature. A fax machine uses a thermal printer. Thermal printer paper tends to darken over time due to exposure to sunlight and heat. The standard of print produced is poor. Thermal printers are widely used in battery powered equipment such as portable calculators.

Direct thermal printers are increasingly replacing the dot matrix printer for printing cash register receipts, both because of the higher print speed and substantially quieter operation. In addition, direct thermal printing offers the advantage of having only one consumable - the paper itself. Thus, the technology is well-suited to unattended applications like gas pumps, information kiosks, and the like.

Until about 2000, most fax machines used direct thermal printing, though, now, only the cheapest models use it, the rest having switched to either thermal wax transfer, laser, or ink jet printing to allow plain-paper printouts. Historically, direct thermal paper has suffered from such limitations as sensitivity to heat, abrasion (the coating can be fragile), friction (which can cause heat, thus darkening the paper), light (causing it to fade), and water. However, more modern thermal coating formulations have resulted in exceptional image stability, with text remaining legible for an estimated 50+ years.

Ink-Jet Printers

Inkjet printers spray very small, precise amounts (usually a few picolitres) of ink onto the media. They are the most common type of computer printer for the general consumer due to their low cost, high quality of output, capability of printing in vivid color, and ease of use. It is the most common printer used with home computers and it can print in either black and white or color.

Compared to earlier consumer-oriented printers, ink jets have a number of advantages. They are quieter in operation than impact dot matrix or daisywheel printers. They can print finer, smoother details through higher print head resolution, and many ink jets with photorealistic-quality color printing are widely available. For color applications including photo printing, ink jet methods are dominant.

Laser Printers

A laser printer is a common type of computer printer that produces high quality printing, and is able to produce both text and graphics. The process is very similar to the type of dry process photocopier first produced by Xerox.

Laser Printers use a laser beam and dry powdered ink to produce a fine dot matrix pattern. This method of printing can generate about 4 pages of A4 paper per minute. The standard of print is very good and laser printers can also produce very good quality printed graphic images too.

SCANNERS:

Technology today is rising to it's heights. For time saving and to have paperless offices we have a need of electronic version of invoice, Material ordering forms, Contract ordering data etc…for filing and database management. Even to automate the process of logging sales data into Excel, a scanner can help one with all of these tasks and more.

A scanner is an optical device that captures images, objects, and documents into a digital format. The image is read as thousands of individual dots, or pixels. It can convert a picture into digital bits of information which are then reassembled by the computer with the help of scanning software. The file of the image can then be enlarged or reduced, stored in a database, or transferred into a word processing or spreadsheet program.

Some of the key considerations for choosing the right scanner for your needs are given below.

a) How you intend to use the scanner?
b) Which type of scanner fits the exact usage?
c) Does one require a Black & White or a Colour quality output?
d) What is the Price and the Software bundles?

Depending upon the usage and the importance of the business if one would like to have quality photographs or other images, than colour quality will be an important characteristic. With both a black and white and a color quality output the bit depth, resolution and dynamic range are essential to selecting the right scanner for ones need.

Scanner Types:

Scanners create a digital reproduction of an image or document and come in a variety of shapes and sizes designed to perform different types of tasks. There are three types of office scanners usually seen in the market and the functions they serve are as follows:

a) Flatbed

The flatbed scanner consists of its own base with a flat piece of glass and cover just as is found on most copiers. The scanning component of flatbeds runs over the length of the image in order to gather data. Flatbeds are useful when a user needs to scan more than single page documents.

Pages from a book, for example, can easily be scanned without having to copy each page individually first.

Scanning objects is also done by flatbeds. By placing a white sheet of paper over a bouquet of flowers a scanner can reproduce what appears to be a stock photo onscreen.

Flatbeds have large footprint and hence take up a lot of desk thus if space is a concern one may go for an alternative.

b) Sheetfed

Sheetfed scanners are only used if one wants to scan for anything other than sheets of paper. The scanning component of a sheetfed is stationary while the document being scanned passes over it's 'eyes' similar to a fax machine. It is so thin just a couple of inches deep, such that it can easily fit between keyboards and monitor.

Sheetfeds usually work best in conjunction with an automatic document feeder for large projects. Pictures and other documents which are smaller than a full page can also be scanned using a sheetfed scanner. They have been known to bend pictures and reproduce less than quality images.

c) Slide

There is a need for accurate reproduce of very small images. For such application the resolution required is very sharp and slide types of scanner create a totally different scanner market. Slides are usually inserted into a tray, much like a CD tray on ones computer, and scanned internally. Most slide scanners can only scan slides, though some newer models can also handle negative strips.

Scanner Uses:

A scanner can do far more than simply scan a photograph, and many of its uses could go a long way to helping a small business. Below are indicated some of the applications for the scanner in a business environment.

1) Graphics

Graphic images are an important part of many businesses specially in marketing and sales functions. Scanners, like digital cameras, enable users to convert photographs, slides, and three-dimensional objects into files that can be pasted into a brochure, inserted into a presentation or posted on the Internet. Using accompanying software, these images can be edited, cropped, or manipulated to fit space and size requirements.

2) Data-Entry

Scanners automatically convert the data into digital files using OCR (Optical Character Recognition) software; this would save time and money which one would pay to someone to manually enter the reams of data into the computer. In conjunction with the software, a scanner reads each page and transfers the text to any number of programs. A form letter can be saved to a word processing program, sales figures to a spreadsheet, even a brochure to web-editing software.

3) Digital-Files

One observes that there are numerous papers filed in three-ring binders or different kinds of manual filing in the offices for records. The process of the manual paper flow can be avoided by using scanners of Digital type. Such scanners can help to create electronic filing cabinets for everything from invoices to expense reports. Forms can be reproduced online, and searchable databases can provide relevant

information in seconds.

Pointer:
A symbol that appears on the display screen and that you move to select objects and commands. Usually, the pointer appears as a small angled arrow. Text -processing applications, however, use an I-beam pointer that is shaped like a capital I.
Pointing device:
A device, such as a mouse or trackball that enables you to select objects on the display screen.
Icons:
Small pictures that represent commands, files, or windows. By moving the pointer to the icon and pressing a mouse button, you can execute a command or convert the icon into a window. You can also move the icons around the display screen as if they were real objects on your desk.

Desktop:
The area on the display screen where icons are grouped is often referred to as the desktop because the icons are intended to represent real objects on a real desktop.
Windows:
You can divide the screen into different areas. In each window, you can run a different program or display a different file. You can move windows around the display screen, and change their shape and size at will.
Menus:
Most graphical user interfaces let you execute commands by selecting a choice from a menu.

In addition to their visual components, graphical user interfaces also make it easier to move data from one application to another. A true GUI includes standard formats for representing text and graphics. Because the formats are well-defined, different programs that run under a common GUI can share data. This makes it possible, for example, to copy a graph created by a spreadsheet program into a document created by a word processor.

Character User Interface/Text User Interface (CUI/TUI)
Short for Character User Interface or Command-line User Interface, CUI is another name for a command line. Early user interfaces were CUI. That is they could only display the characters defined in the ASCII set. Examples of this type of interface are the command line interfaces provided with DOS 3.3 and early implementations of UNIX and VMS.

This was limiting, but it was the only choice primarily because of 2 hardware constraints. Early CPUs did not have the processing power to manage a GUI. Also, the video controllers and monitors were unable to display the high resolution necessary to implement a GUI.

FUNCTIONS OF CPU:

Process Management

The CPU executes a large number of programs. While its main concern is the execution of user programs, the CPU is also needed for other system activities. These activities are called processes. A process is a program in execution. Typically, a batch job is a process. A time-shared user program is a process. A system task, such as spooling, is also a process. For now, a process may be considered as a job or a time-shared program, but the concept is actually more general.

In general, a process will need certain resources such as CPU time, memory, files, I/O devices, etc., to accomplish its task. These resources are given to the process when it is created. In addition to the various physical and logical resources that a process obtains when it is created, some initialization data (input) may be passed along.

We emphasize that a program by itself is not a process; a program is a passive entity. It is known that two

processes may be associated with the same program; they are nevertheless considered two separate execution sequences.

The operating system is responsible for the following activities in connection with processes managed.

- The creation and deletion of both user and system processes
- The suspension and resumption of processes.
- The provision of mechanisms for process synchronization
- The provision of mechanisms for deadlock handling.

Memory Management

Memory is central to the operation of a modern computer system. Memory is a large array of words or bytes, each with its own address. Interaction is achieved through a sequence of reads or writes of specific memory address. The CPU fetches from and stores in memory. In order for a program to be executed it must be mapped to absolute addresses and loaded in to memory.

In order to improve both the utilization of CPU and the speed of the computer's response to its users, several processes must be kept in memory.

The operating system is responsible for the following activities in connection with memory management.

- Keep track of which parts of memory are currently being used and by whom.
- Decide which processes are to be loaded into memory when memory space becomes available.
- Allocate and de-allocate memory space as needed.

Secondary Storage Management

The main purpose of a computer system is to execute programs. These programs, together with the data they access, must be in main memory during execution. Since the main memory is too small to permanently accommodate all data and program, the computer system must provide secondary storage to backup main memory. Most modem computer systems use disks as the primary on-line storage of information, of both programs and data.

Most programs, like compilers, assemblers, sort routines, editors, formatters, and so on, are stored on the disk until loaded into memory, and then use the disk as both the source and destination of their processing. Hence the proper management of disk storage is of central importance to a computer system.

There are few alternatives. Magnetic tape systems are generally too slow. In addition, they are limited to sequential access. Thus tapes are more suited for storing infrequently used files, where speed is not a primary concern.

The operating system is responsible for the following activities in connection with disk management

- Free space management
- Storage allocation
- Disk scheduling.

Input Output System

One of the purposes of an operating system is to hide the peculiarities of specific hardware devices from the user. For example, in UNIX, the peculiarities of Input/Output devices are hidden from the bulk of the operating system itself by the INPUT/OUTPUT system. The Input/Output system consists of:

- A buffer caching system
- A general device driver code
- Drivers for specific hardware devices.

Only the device driver knows the peculiarities of a specific device.

File Management

File management is one of the most visible services of an operating system. Computers can store information in several different physical forms; magnetic tape, disk, and drum are the most common forms. Each of these devices has it own characteristics and physical organization. For convenient use of the computer system, the operating system provides a uniform logical view of information storage. The operating system abstracts from the physical properties of its storage devices to define a logical storage unit, the file. Files are mapped, by the operating system, onto physical devices.

A file is a collection of related information defined by its creator. Commonly, files represent programs (both source and object forms) and data. Data files may be numeric, alphabetic or alphanumeric. Files may be free-form, such as text files, or may be rigidly formatted. In general files are a sequence of bits, bytes, lines or records whose meaning is defined by its creator and user. It is a very general concept.
The operating system implements the abstract concept of the file by managing mass storage device, such as tapes and disks. Also files are normally organized into directories to ease their use. Finally, when multiple users have access to files, it may be desirable to control by whom and in what ways files may be accessed.

The operating system is responsible for the following activities in connection with file management:

- The creation and deletion of files
- The creation and deletion of directory
- The support of primitives for manipulating files and directories
- The mapping of files onto disk storage.
- Backup of files on stable (non volatile) storage.

Protection System

The various processes in an operating system must be protected from each other's activities. For that purpose, various mechanisms which can be used to ensure that the files, memory segment, CPU and other resources can be operated on only by those processes that have gained proper authorization from the operating system.

Protection refers to a mechanism for controlling the access of programs, processes, or users to the resources defined by a computer controls to be imposed, together with some means of enforcement. An unprotected resource cannot defend against use (or misuse) by an unauthorized or incompetent user.

Networking

A distributed system is a collection of processors that do not share memory or a clock. Instead, each processor has its own local memory, and the processors communicate with each other through various communication lines, such as high speed buses or telephone lines. Distributed systems vary in size and function. They may involve microprocessors, workstations, minicomputers, and large general purpose computer systems.

The processors in the system are connected through a communication network, which can be configured in the number of different ways. The network may be fully or partially connected. The communication network design must consider routing and connection strategies, and the problems of connection and security. A distributed system provides the user with access to the various resources the system maintains. Access to a shared resource allows computation speed-up, data availability, and reliability.

Command Interpreter System

One of the most important components of an operating system is its command interpreter. The command interpreter is the primary interface between the user and the rest of the system. Many commands are given to the operating system by control statements. When a new job is started in a batch system or when a user logs-in to a time-shared system, a program which reads and interprets control statements is automatically executed.

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 $10^0 = 1$ that is $\mathbf{10^3\ 10^2\ 10^1\ 10^0}$

For fractional numbers, the weights are negative powers of ten that decrease from left to right beginning with 10^{-1} that is $\mathbf{10^2\ 10^1\ 10^0.\ 10^{-1}\ 10^{-2}\ 10^{-3}}$
The value of a decimal number is the sum of digits after each digit has been multiplied by its weights as in following examples

Express the decimal number 87 as a sum of the values of each digit.

The digit 8 has a weight of 10 which is 10 as indicated by its position. The digit 7 has a weight of 1 which is 10^0 as indicated by its position.

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

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

$$725.45 = (7 \times 10^{2}) + (2 \times 10^{1}) + (5 \times 10^{o}) + (4 \times 10^{-1}) + (5 \times 10^{-2}) = 700 + 20 + 5 + 0.4 + 0.05$$

BINARY NUMBERS

The binary system is less complicated than the decimal system because it has only two digits, it is a base-two 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:

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

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 **- 1**. Largest decimal number = **- 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

PLANNING THE COMPUTER PROGRAM

The Programming Process - Purpose

- **Understand the problem**
  - Read the problem statement
  - Question users
  - Inputs required
  - Outputs required
  - Special formulas
  - Talk to users
- **Plan the logic**
  - Visual Design Tools Input
    - record chart Printer
    - spacing chart Hierarchy
    - chart
    - Flowchart
  - Verbal Design Tools
    - Narrative Description

- Pseudocode

- **Code the program**
  - Select an appropriate programming language
  - Convert flowchart and/or Pseudocode instructions into programming language statements

- **Test the program**
  - Syntax errors
  - Runtime errors
  - Logic errors
  - Test Data Set

- **Implement the program**
  - Buy hardware
  - Publish software
  - Train users
  - Implementation Styles
    - Crash
    - Pilot
    - Phased
    - Dual

- **Maintain the program**
  - Maintenance programmers
  - Legacy systems
  - Up to 85% of IT department budget

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 leading to software 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

LIMITATIONS OF USING FLOWCHARTS

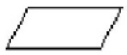
1. Complex logic: Sometimes, the program logic is quite complicated. In that case, flowchart becomes complex and clumsy.
2. Alterations and Modifications: If alterations are required the flowchart may require re-drawing completely.
3. Reproduction: As the flowchart symbols cannot be typed, reproduction 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

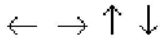
Connector or joining of two parts of program

Magnetic Tape

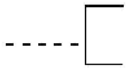
Magnetic Disk

Off-page connector

Flow line



Annotation

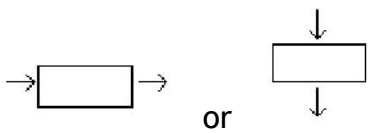


Display

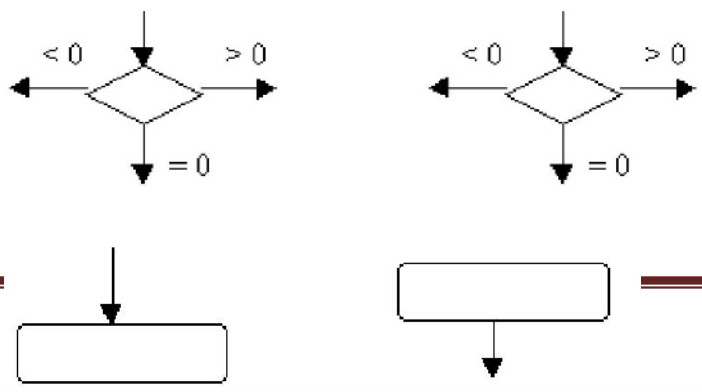
Flowchart Symbols

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.

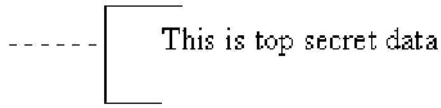


e. Only one flow line should enter a decision symbol, but two or three flow lines, one for each possible answer, should leave the decision symbol.



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.

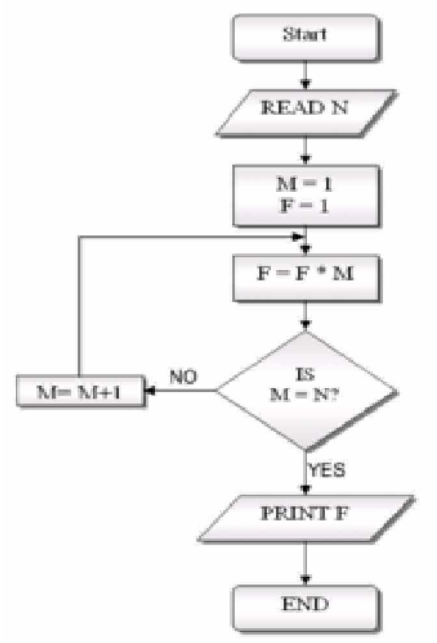


h. If the flowchart becomes complex, it is better to use connector symbols to reduce the number of flow lines. Avoid the intersection of flow lines if you want to make it more effective and better way of communication.
i. Ensure that the flowchart has a logical *start* and *finish*.
j. It is useful to 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
```

UNIT II C PROGRAMMING BASICS 10

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 andBranching – Looping statements – solving simple scientific and statistical problems

OVERVIEW OF 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");
}
```

A NOTE ABOUT C PROGRAMS

In C, lowercase and uppercase characters are very important! All commands in C must be lowercase. The C programs starting point is identified by the word

main()

This informs the computer as to where the program actually starts. The brackets that follow the keyword *main* indicate that there are no arguments supplied to this program (this will be examined later on).

The two braces, { and }, signify the begin and end segments of the program. The purpose of the statment

include <stdio.h>

is to allow the use of the *printf* statement to provide program output. Text to be displayed by *printf()* must be enclosed in double quotes. The program has only one statement

printf("Programming in C is easy.\n");

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. Thus the program prints

Programming in C is easy.

and the cursor is set to the beginning of the next line. As we shall see later on, what follows the \ character will determine what is printed, ie, a tab, clear screen, clear line etc. Another important thing to remember is that all C statements are terminated by a semi-colon ;

Summary of major points:

- ν program execution begins at *main()*
- ν keywords are written in lower-case
- ν statements are terminated with a semi-colon
- ν text strings are enclosed in double quotes
- ν C is case sensitive, use lower-case and try not to capitalise variable names
- ν \n means position the cursor on the beginning of the next line
- ν *printf()* can be used to display text to the screen
- ν The curly braces {} define the beginning and end of a program block.

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                    */

int main()
```

```
{

}
```

VARIABLE

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 AND CONSTANTS

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';
```

Note the assignment of the character *A* to the variable *letter* is done by enclosing the value in **single quotes**. Remember the golden rule: Single character - Use single quotes.

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 io n is a sequence of operators and operands that specifies computation of avalue, 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.

n	*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()
{
        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;
else if( condition 2 ) statement2;
else if( condition 3 ) 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;
        else if(operator == '/')
                result = number1 / number2;
        else if(operator == '+')
                result = number1 + number2;
        else if(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 is executed if the cases are not met. The right brace at the end signifies the end of the case selections.

Example:

```
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("Invalid option selected\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 9

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

INTRODUCTION

An array is a group of related data items that share a common name. For instance, we can define array name **salary** to represent a set of salary of a group of employees. A particular value is indicated by writing a number called index number or subscript in brackets after the array name.

Eg: **salary[10]**

ONE DIMENSIONAL ARRAY

An array with a single subscript is known as one dimensional array.

Eg: 1) **int number[5];**

The values to array elements can be assigned as follows.

Eg: 1) **number[0] = 35;**
number[1] = 40;
number[2] = 20;

Declaration of Arrays

The general form of array declaration is

type variable-name[size];

The type specifies the type of element that will be contained in the array, such as **int, float,** or **char** and the size indicates the maximum number of elements that can be stored inside the array.

Eg: **1) float height[50];**
2) int group[10];
3)char name[10];

Initialization of Arrays

The general form of initialization of arrays is:

static type array-name[size] = {list of values};

Eg:1) **static int number[3] = {0,0};**

If the number of values in the list is less than the number of elements, then only that many elements will be initialized. The remaining elements will be set to zero

automatically. Initialization of arrays in C suffers two drawbacks

- There is no convenient way to initialize only selected elements.
- There is no shortcut method for initializing a large number of array elements like the one available in FORTRAN.

We can use the word 'static' before type declaration. This declares the variable as a static variable.

Eg : 1) **static int counter[] = {1,1,1};**

2)
```
.....
.....
for(i =0; i < 100; i = i+1)
{
 if i < 50
    sum[i] = 0.0;
 else
    sum[i] = 1.0;
}
.....
.....
```

Program

```
/*Program showing one-dimensional array*/
main()
{
int i;
float x[10],value,total; printf("Enter
10 real numbers:\n"); for(i =0; i < 10;
i++)
{
scanf("%f",&value);
x[i] = value;
}
total = 0.0;

for(i = 0; i < 10; i++)
total = total + x[i] * x[i];
printf("\n");
for(i = 0; i < 10; i++) printf("x[%2d]
= %5.2f \n",i+1,x[i]); printf("\nTotal
= %.2f\n",total);
```

```
}
```

OUTPUT

```
Enter 10 real numbers:
1.1 2.2 3.3 4.4  5.5 6.6 7.7 8.8 9.9 10.10

x[1] = 1.10 x[2]
= 2.20 x[3] =
3.30 x[4] = 4.40




x[5] = 5.50 x[6]
= 6.60 x[7] =
7.70 x[8] = 8.80
x[9] = 9.90
x[10] = 10.10
Total = 446.86
```

TWO-DIMENSIONAL ARRAYS

Two-dimensional arrays are declared as follows

type array- name[row_size][column_size];

Eg: **product[i][j] = row * column;**

Program

```
/*Program to print multiplication table*/
#define    ROWS              5
#define     COLUMNS       5
main()
{
int   row, column, product[ROWS][COLUMNS];
int i, j;
printf("Multiplication table\n\n:") ;
printf("  ");
for(j = 1; j <= COLUMNS; j++)
printf("%4d",j);
printf("\n");
printf("                                \n");
for(i = 0; i < ROWS; i++)
{
row = i + 1;
printf("%2d|", row);
for(j =1; j <= COLUMNS; j++)
{
column = j;
product[i][j] = row * column;
```

```
printf("%4d", product[i][j]);
}
printf("\n");
}
}
```

OUTPUT

```
Multiplication Table
1    2   3   4   5
```


Check Your Progress

Ex 1) Give examples for one dimensional array.

Ex 2) Give examples for two dimensional array.

MULTIDIMENSIONAL ARRAY

C allows arrays of three or more dimensions. The exact limit is determined by the compiler. The general form of a multidimensional array is

type **array_name**[s1][s2][s3]…s[m];

Eg: 1. **int survey[3][5][12];**
2. **float table[5][4][5][3];**

HANDLING OF CHARACTER STRINGS

INTRODUCTION

A string is a array of characters. Any group of characters(except the double quote sign) defined between double quotation marks is a constant string.
Eg: 1) "Man is obviously made to think"

If we want to include a double quote in a string, then we may use it with the back slash.

Eg: **printf("\"well done!\"");**

will output

"well done!"

The operations that are performed on character strings are

- Reading and writing strings.
- Combining strings together.
- Copying one string to another.
- Comparing strings for equality.
- Extracting a portion of a string.

DECLARING AND INITIALIZING STRING VARIABLES

A string variable is any valid C variable name and is always declared as an array. The general form of declaration of a string variable is

char string_name[size];

Eg: **char city[10];**

char name[30];

When the compiler assigns a character string to a character array, it automatically supplies a null character ('\0') at the end of the string. Therefore, the size should be equal to the maximum number of characters in the string plus one. C permits a character array to be initialized in either of the following two forms

static char city[9] = "NEW YORK";

Reading Words

The familiar input function scanf can be used with %s format specification to read in a string of characters.

Eg: **char address[15];**
scanf("%s",address);

Program

```
/*Reading a series of words using scanf
function*/ main()
{
char word1[40],word2[40],word3[40],word4[40];
printf("Enter text:\n");
scanf("%s %s",word1,
word2); scanf("%s", word3);
scanf("%s",word4);
printf("\n");
printf("word1 = %s \n word2 = %s \n",word1, word2);
printf("word3 = %s \n word4 = %s \n",word3, word4);
}
```

OUTPUT
Enter text:
Oxford Road, London M17ED
Word1 = Oxford
Word2 = Road
Word3 = London
Word4 = M17ED
Note: Scanf function terminates its input on the first white space it finds.

Reading a Line of Text

It is not possible to use scanf function to read a line containing more than one word. This is because the scanf terminates reading as soon as a space is encountered in the input. We can use the getchar function repeatedly to read single character from the terminal, using the function **getchar**. Thus an entire line of text can be read and stored in an array.

Program

```
/*Program to read a line of text from terminal*/
#include<stdio.h>
main()
{
char line[81],character;
int c;
c = 0;
printf("Enter text. Press<Return>at end
\n"); do
{
character = getchar();

        line[c] = character; c++;

}
while(character !=
'\n'); c = c-1;
line[c] = '\0';
printf("\n %s \n",line);
}
```

OUTPUT

```
Enter text. Press<Return>at end
Programming in C is interesting
Programming in C is interesting
```

WRITING STRINGS TO SCREEN

We have used extensively the printf function with %s format to print strings to the screen. The format %s can be used to display an array of characters that is terminated by the null character.

For eg, the statement

printf("%s", name);

can be used to display the entire contents of the array **name**.

ARITHMETIC OPERATIONS ON CHARACTERS

C allows us to manipulate characters the same way we do with numbers. Whenever a character constant or character variable is used in an expression, it is automatically converted into integer value by the system.

For eg, if the machine uses the ASCII representation, then,

```
x = 'a';
printf("%d \n",x);
```

will display the number 97 on the screen.

The C library supports a function that converts a string of digits into their integer values. The function takes the form

x = atoi(string)

PUTTING STRINGS TOGETHER

Just as we cannot assign one string to another directly, we cannot join two strings together by the simple arithmetic addition. That is, the statements such as

```
string3 = string1 + string2;
string2 = string1 + "hello";
```

are **not** valid. The characters from string1 and string2 should be copied into string3 one after the other. The process of combining two strings together is called concatenation.

COMPARISON OF TWO STRINGS

C does not permit the comparison of two strings directly. That is, the statements such as

```
if(name1 == name2)
if(name == "ABC");
```

are **not** permitted. It is therefore necessary to compare the two strings to be tested, character by character. The comparison is done until there is a mismatch or one of the strings terminate into a null character, whichever occurs first.

STRING - HANDLING FUNCTIONS

C library supports a large number of string- handling functions that can be used to carry out many of the string manipulation activities. Following are the most commonly used string-handling functions.

Function	Action
strcat( )	Concatenates two strings
strcmp( )	Compares two strings
strcpy( )	Copies one string over another
strlen( )	Finds the length of the string

strcat( ) Function

The strcat function joins two strings together. It takes the following form

strcat(string1,string2**);**

Eg: **strcat(part1, "GOOD");**

strcat(strcat(string1,string2),string3);

Here three strings are concatenated and the result is stored in string1.

strcmp( ) Function

It is used to compare two strings identified by the arguments and has a value 0 if they are equal.It takes the form:

strcmp(string1,string2);

Eg: 1) **strcmp(name1,name2);**
2) strcmp(name1,"john";
3) strcmp("ram", "rom");

strcpy( ) Function

This function works almost like a string assignment operator. It takes the form

strcpy(string1,string2);

This assigns the content of string2 to string1.

Eg: 1) **strcpy(city, "DELHI");**
2) strcpy(city1,city2);

strlen( ) Function

This function counts and returns the number of characters in a string.

n = **strlen**(string);

Program

```
/*Illustration of string- handling functions*/
#include<string.h>
main()
{
char s1[20],s2[20],s3[20];
int x, l1, l2, l3;
```

```
printf("Enter two string constants \n");
printf("?");
scanf("%s %s", s1, s2);
x = strcmp(s1, s2);
if(x != 0)
printf("Strings are not equal \n");
strcat(s1, s2);

else
printf("Strings are equal \n");
strcpy(s3,s1);
l1 = strlen(s1);
l2 = strlen(s2);
l3 = strlen(s3);
printf("\ns1 = %s \t length = %d characters \n",s1, l1);
printf("\ns2= %s \t length = %d characters \n",s2, l2);
printf("\ns3 = %s \t length = %d characters \n",s3, l3);
}
```

OUTPUT

```
Enter two string constants

? New York
Strings are not equal
s1 = New York          length = 7 characters
s2 = York              length = 4 characters
s3 = New York          length = 7 characters


Enter two string constants
? London London
Strings are equal
s1 = London            length = 6 characters
s2 = London            length = 6 characters
s3 = London            length = 6 characters
```

UNIT IV FUNCTIONS AND POINTERS 9

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

FUNCTIONS

A function in C can perform a particular task, and supports the concept of modular programming design techniques.

We have already been exposed to functions. The main body of a C program, identified by the keyword *main*, and enclosed by the left and right braces is a function. It is called by the operating system when the program is loaded, and when terminated, returns to the operating system.

Functions have a basic structure. Their format is

```
return_data_type function_name ( arguments, arguments )
data_type_declarations_of_arguments;
{
        function_body
}
```

It is worth noting that a return_data_type is assumed to be type *int* unless otherwise specified, thus the programs we have seen so far imply that *main()* returns an integer to the operating system.

```
return_data_type function_name (data_type variable_name, data_type variable_name, .. )
{
        function_body
}
```

simple function is,

```
void print_message( void )
{
        printf("This is a module called print_message.\n");
}
```

Example:

Now lets incorporate this function into a program.

```
/* Program illustrating a simple function call */
#include <stdio.h>
```

```
void print_message( void );   /* ANSI C function prototype */

void print_message( void )   /* the function code */
{
        printf("This is a module called print_message.\n");
}

main()
{
        print_message();
}
```

To call a function, it is only necessary to write its name. The code associated with the function name is executed at that point in the program. When the function terminates, execution begins with the statement which follows the function name.

In the above program, execution begins at *main()*. The only statement inside the main body of the program is a call to the code of function *print_message()*. This code is executed, and when finished returns back to *main()*.

As there is no further statements inside the main body, the program terminates by returning to the operating system.

RETURNING FUNCTION RESULTS

This is done by the use of the keyword *return*, followed by a data variable or constant value, the data type of which must match that of the declared return_data_type for the function.

```
float add_numbers( float n1, float n2 )
{
        return n1 + n2;   /* legal */
        return 6;         /* illegal, not the same data type */
        return 6.0;       /* legal */
}
```

It is possible for a function to have multiple return statements.

```
int validate_input( char command )

        switch( command ) { case '+' :

                case '-' : return 1;
                case '*' :
                case '/' : return 2;
                default : return 0;
        }
}
```

Example:

```
/* Simple multiply program using argument passing */
        #include <stdio.h>

        int calc_result( int, int );          /* ANSI function prototype */
```

```
int calc_result( int numb1, int numb2 )
{
        auto int result;
        result = numb1 * numb2;
        return result;
}

main()
{
        int digit1 = 10, digit2 = 30, answer = 0;
        answer = calc_result( digit1, digit2 );
        printf("%d multiplied by %d is %d\n", digit1, digit2, answer );
}
```

RECURSION

This is where a function repeatedly calls itself to perform calculations. Typical applications are games and Sorting trees and lists.

Consider the calculation of 6! (6 factorial)

ie 6! = 6 * 5 * 4 * 3 * 2 * 1
6! = 6 * 5!
6! = 6 * (6 - 1)!
n! = n * (n - 1)!

Example:

```
/* example for demonstrating recursion */
#include <stdio.h>

long int factorial( long int );        /* function prototype */

long int  factorial( long int n )
{
        long int result;

        if( n == 0L )
                result = 1L;
        else
                result = n * factorial( n - 1L ); return
        ( result );
}

main()
{
        int j;

        for( j = 0; j < 11; ++j )
                printf("%2d! = %ld\n", factorial( (long) j) );
}
```

CALL 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

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 BYREFERENCE

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

INTRODUCTION

Pointers are another important feature of C language. Although they may appear a little confusing for a beginner, they are powerful tool and handy to use once they are mastered. There are a number of reasons for using pointers.

1. A pointer enables us to access a variable that is defined outside the function.
2. Pointers are more efficient in handling the data tables.
3. Pointers reduce the length and complexity of a program.
4. They increase the execution speed.
5. The use of a pointer array to character strings result in saving of data storage space in memory.

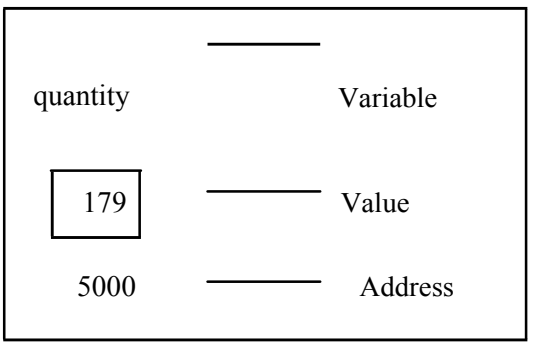
UNDERSTANDING POINTERS

Whenever we declare a variable, the system allocates, somewhere in the memory, an appropriate location to hold the value of the variable. Since, every byte has a unique address number, this location will have its own address number.

Consider the following statement:

int quantity = 179;

This statement instructs the system to find a location for the integer variable quantity and puts the value 179 in that location. Assume that the system has chosen the address location 5000 for quantity. We may represent this as shown below.



Representation of a variable

During execution of the program, the system always associates the name quantity with the address 5000. To access the value 179 we use either the name quantity or the address 5000. Since memory addresses are simply numbers, they can be assigned to some variables which can be stored in memory, like any other variable. **Such variables that hold memory addresses are called pointers. A pointer is, therefore, nothing but a variable that contains an address which is a location of another variable in memory.**

Since a pointer is a variable, its value is also stored in the memory in another location. Suppose, we assign the address of quantity to a variable p. The link between the variables p and quantity can be visualized as shown below. The address of p is 5048.

Variable	*Value*	*Address*
quantity	179	5000
p	5000	5048

Pointer as a variable

Since the value of the variable p is the address of the variable quantity, we may access the value of quantity by using the value of p and therefore, we say that the variable p 'points' to the variable quantity. Thus, p gets the name 'pointer'.

CESSING THE ADDRESS OF A VARIABLE

The actual location of a variable in the memory is system dependent and therefore, the address of a variable is not known to us immediately. However we determine the address of a variable by using the operand & available in C. The operator immediately preceding the variable returns the address of the variable associated with it. For example, the statement

p = &quantity;

would assign the address 5000(the location of quantity) to the variable p. The &operator can be remembered as 'address of'.

The & operator can be only used with a simple variable or an array element. The following are **illegal** use of address operator:

1. &125 (pointing at constants).
2. int x[10];
   &x (pointing at array names).
3. &(x+y) (pointing at expressions).

If x is an array ,then expressions such as

&x[0] and &x[i + 3]

are **valid** and represent the addresses of 0^{th} and $(i + 3)^{th}$ elements of x.

The program shown below declares and initializes four variables and then prints out these

values with their respective storage locations.

Program

```
/*****************************************************************/ /*
ACCESSING ADDRESSES OF VARIABLES */
/*****************************************************************/
main()
{
        char a; int
        x; float p,
        q;
         a = 'A';
         x = 125;
         p = 10.25 , q = 18.76;
         printf("%c  is stored as addr %u . \n", a, &a);
         printf("%d  is stored as addr %u . \n",x , &x);
         printf("%f  is stored as addr %u . \n", p, &p);
         printf("%f  is stored as addr %u . \n", q, &q);
}
```

A is stored at addr 44336
125 is stored at addr 4434
10.250000 is stored at addr 4442
18.760000 is stored at addr 4438.

DECLARING AND INITIALIZING POINTERS

Pointer variables contain addresses that belong to a separate data type, which must be declared as pointers before we use them. The declaration of the pointer variable takes the following form:

data type *pt _name;

This tells the compiler three things about the variable pt _name:

1. The asterisk(*) tells that the variable pt _name.
2. pt _name needs a memory location.
3. pt_name points to a variable of type data type.

Example:

```
int *p;
float *x;
```

Once a pointer variable has been declared, it can be made to point to a variable using an assignment operator such as

p= &quantity;

Before a pointer is initialized it should not be used.

Ensure that the pointer variables always point to the corresponding type of data.

Example:

```
float a, b;
int  x,  *p;
p = &a;
b = *p;
```

will result in erroneous output because we are trying to assign the address of a float variable to an integer pointer. When we declare a pointer to be of int type, the system assumes that any address that a pointer will hold will point to an integer variable.

Assigning an absolute address to a pointer variable is prohibited. The following is wrong.

```
int *ptr;
….
ptr = 5368;
….
….
```

A pointer variable can be initialized in its declaration itself. For example,

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

is perfectly valid. It declares x as an integer variable and p as a pointer variable and then initializes p to the address of x. The statement

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

is not valid.

ACCESSING A VARIABLE THROUGH ITS POINTER

To access the value of the variable using the pointer, another unary operator *(asterisk), usually known as the indirection operator is used. Consider the following statements:

```
int quantity, *p, n;
quantity = 179;
p= &quantity;
```

n= *p;

The statement n=*p contains the indirection operator *. When the operator * is placed before a pointer variable in an expression (on the right-hand side of the equal sign), the pointer returns the value of the variable of which the pointer value is the address. In this case, *p returns the value of the variable quantity, because p is the address of the quantity. The * can be remembered as 'value at address'. Thus the value of n would be 179. The two statements

p= &quantity;

n= *p; are equivalent to

n= *&quantity;

which in turn is equivalent to

n= quantity;

The following program illustrates the distinction between pointer value and the value it points to and **the use of indirection operator(*) to access the value pointed to by a pointer.**

Check Your Progress

Ex 1) Specify a few reasons to use Pointers.

Ex 2) Pointer variable stores _____

Program ACCESSING VARIABLES USING POINTERS

```
main( )
{
        int x, y ;
        int * ptr;
        x =10;
        ptr = &x;
        y = *ptr;
        printf ("Value of x is %d \n\n",x);
       printf ("%d is stored at addr %u \n" , x, &x);
       printf ("%d is stored at addr %u \n" , *&x, &x);
       printf ("%d is stored at addr %u \n" , *ptr, ptr);
       printf ("%d is stored at addr %u \n" , y, &*ptr);
       printf ("%d is stored at addr %u \n" , ptr, &ptr);
       printf ("%d is stored at addr %u \n" , y, &y);
        *ptr= 25;
        printf("\n Now x = %d \n",x);
  }
```

The statement ptr = &x assigns the address of x to ptr and y = *ptr assigns the value

pointed to by the pointer ptr to y.

Note the use of assignment statement

*ptr=25;

This statement puts the value of 25 at a memory location whose address is the value of ptr. We know that the value of ptr is the address of x and therefore the old value of x is replaced by 25. This, in effect, is equivalent to assigning 25 to x. This shows how we can change the value of a variable indirectly using a pointer and the indirection operator.

POINTER EXPRESSIONS

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.

1) y = *p1* *p2; same as (* p1) * (* p2)

2) sum = sum + *p1;

3) z = 5* - *p2/ *p1; same as (5 * (-(* p2)))/(* p1)

4) *p2 = *p2 + 10;

Note that there is a blank space between / and * in the statement 3 above.

C allows us to add integers to or subtract integers from pointers , as well as to subtract one pointer from another. p1 + 4, p2 – 2 and p1 – p2 are all allowed. If p1 and p2 are both

pointers to the same array, then p2 – p1 gives the number of elements between p1 and p2. We may also use short-hand operators with the pointers.

p1++;

--p2;

Sum += *p2;

Pointers can also be compared using the relational operators. Pointers cannot be used in division or multiplication. Similarly two pointers cannot be added.

A program to illustrate the use of pointers in arithmetic operations.

Program POINTER EXPRESSIONS

```
main ( )
{
        int a, b, *p1,* p2, x, y, z;
        a  = 12;
        b  = 4;
        p1 = &a;
        p2 = &b;
        x  =  *p1 *  *p2 – 6;
        y  =  4* - *p2 / *p1 + 10;



        printf("Address of a = %u\n", p1);

        printf("Address of b = %u\n", p2);
        printf("\n");
        printf("a = %d, b = %d\n", a, b);
        printf("x = %d, y = %d\n", x, y);
        *p2 = *p2 + 3;
        *p1 =  *p2 - 5;
         z    =  *p1 * *p2 – 6;
        printf("\n a = %d, b = %d," , a ,
        b); printf("\n z = %d\n" , z);
}
```

POINTER INCREMENTS AND SCALE FACTOR

We have seen that the pointers can be incremented

like p1 = p2 + 2;

p1 = p1 + 1;

and so on .

Remember, however, an expression

like p1++;

will cause the pointer p1 to point to the next value of its type.

That is, when we increment a pointer, its value is increased by the length of the data type that it points to. This length is called the scale factor.

The number of bytes used to store various data types depends on the system and can be found by making use of size of operator. For example, if x is a variable, then size of(x) returns

the number of bytes needed for the variable.

POINTERS AND ARRAYS

When an array is declared, the compiler allocates a base address and sufficient amount of storage to contain all the elements of array in contiguous memory location. The base address is the location of the first element (index 0) of the array. The compiler also defines the array name as a constant pointer to the first element. Suppose we declare an array x as follows:

static int x[5] = {1,2,3,4,5};

Suppose the base address of x is 1000and assuming that each integer requires two bytes, the five elements will be stored as follows:

Elements	x[0]	x[1]	x[2]	x[3]	x[4]
Value	1	2	3	4	5
Address	1000	1002	1004	1006	1008

The name x is defined as a constant pointer pointing to the first element x[0] and therefore value of x is 1000, the location where x[0] is stored . That is ,

x = &x[0] =1000

Accessing array elements using the pointer

Pointers can be used to manipulate two-dimensional array as well. An element in a two-dimensional array can be represented by the pointer expression as follows:

((a+i)+j) or *(*(p+i)+j)

The base address of the array a is &a[0][0] and starting at this address, the compiler allocates contiguous space for all the elements, row-wise. That is, the first element of the second row is placed immediately after the last element of the first row, and so on.

A program using Pointers to compute the sum of all elements stored in an array is presented below:

POINTERS IN ONE-DIMENSONAL ARRAY

```
main ( )
{
        int *p, sum , i
        static int x[5] = {5,9,6,3,7};
        i = 0; p =
        x; sum =
        0;
        printf("Element Value Address \n\n");
        while(i < 5)
        {
           printf(" x[%d} %d %u\n", i, *p, p);
           sum = sum + *p;
           i++, p++;
        }
        printf("\n Sum = %d \n", sum);
        printf("\n &x[0] = %u \n", &x[0]);
        printf("\n p = %u \n", p);
        }
```

Output

```
Element      Value   Address
   X[0]        5        166
   X[1]        9        168
   X[2]        6        170
   X[3]        3        172
   X[4]        7        174
   Sum     = 55
   &x[0] = 166
    p       = 176
```

POINTERS AND CHARACTER STRINGS

We know that a string is an array of characters, terminated with a null character. Like in one-dimensional arrays, we can use a pointer to access the individual characters in a string. This is illustrated in the program given below.

```
/* Pointers and character Strings */
main()
  {
     char * name;
     int length;
    char * cptr = name;
```

```
        name = "DELHI";
        while ( *cptr != '\0')
        {
          printf( "%c is stored at address %u \n", *cptr,cptr);
          cptr++;
        }

        length = cptr-name;
        printf("\n length = %d \n", length);
}
```

String handling by pointers

One important use of pointers in handling of a table of strings. Consider the following array of strings:

```
char name[3][25];
```

This says that name is a table containing three names, each with a maximum length of 25 characters (including null character).

Total storage requirements for the name table are 75 bytes.

Instead of making each row a fixed number of characters , we can make it a pointer to a string of varying length.

For example,

```
static char *name[3] = { "New zealand",
                         "Australia",
                         "India"
                        };
```

declares name to be an array of three pointers to characters, each pointer pointing to a particular name as shown below:

name[0] → New Zealand

name[1] → Australia

name[2] → India

POINTERS AS FUNCTION ARGUMENTS

Program POINTERS AS FUNCTION PARAMETERS

```
main ( )
{
        int x , y;
        x = 100;
        y = 200;
        printf("Before exchange : x = %d y = %d \n\n ", x , y);
        exchange(&x, &y);
        printf("After exchange : x = %d y = %d \n\n " , x , y);
}
exchange(a, b)
int *a, *b;
{
        int t;
        t = * a;        /*Assign the value at address a to t*/
        * a = * b ;     /*Put the value at b into a*/
        * b = t;        /*Put t into b*/
}
```

In the above example, we can pass the address of the variable a as an argument to a function in the normal fashion. The parameters receiving the addresses should be pointers. The process of calling a function using pointers to pass the addresses of variable is known as call by reference. The function which is called by 'Reference' can change the value of the variable used in the call.

Passing of pointers as function parameters

1. The function parameters are declared as pointers.
2. The dereference pointers are used in the function body.
3. When the function is called, the addresses are passed as actual arguments. Pointers parameters are commonly employed in string functions.

Pointers to functions

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

```
type (*fptr)( );
```

This tells the compiler that fptr is a pointer to a function which returns type value.

A program to illustrate a function pointer as a function argument.

Program

POINTERS TO FUNCTIONS

```
#include <math.h>
#define PI 3.141592
main ( )
{
        double y( ), cos( ), table( );
        printf("Tableof y(x) = 2*x*x-
        x+1\n\n"); table(y, 0.0 , 2.0, 0.5);

        printf("\n Table of cos(x) \n\n");
        table(cos, 0.0, PI , 0.5);
}
        double table(f, min, max, step)
        double (*f) ( ), min, max , step;
        {
           double a, value;
            for( a = min; a < = max; a + = step)
            {
                value = (*f)(a);
                printf("%5.2f %10.4f\n", a, value);
             }
        }
        double y(x)
        double x;
        {
                return (2*x*x-x+1);
        }
```

POITNTERS AND STRUCTERS

The name of an array stands for the address of its zeroth element. Consider the following declaration:

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

This statement declares product as an array of two elements, each of type of struct inventory and ptr as a pointer to data objects of the type struct inventory.

The assignment

ptr = product;

would assign the address of the zeroth element of product to ptr. Its members can be accessed using the following notation .

ptr → name

ptr → number

ptr → price

Initially the pointer ptr will point to product[0], when the pointer ptr is incremented by one it will point to next record, that is product[1].

We can also use the notation

(*ptr).number

to access the member number.

A program to illustrate the use of structure pointers.

```
Program POINTERS TO STRUCTURE VARIABLES
struct invent
{
        char  *name[20];
        int number;  float
        price;
};
main( )
{
        struct invent product[3],
        *ptr; printf("INPUT\n\n");
        for(ptr = product; ptr < product + 3; ptr + +)

                scanf("%s %d %f", ptr → name, &ptr → number , & ptr →
        price);
        printf("\Noutput\n\n");
        ptr = product; while(ptr
        < product +3)

        {
        printf("%-20s %5d %10.2f\n" ,
                        ptr →  name,
                       ptr  →  number ,
                        ptr →  price);  ptr++;
          }
}
```

While using structure pointers we should take care of the precedence of operators.

For example, given the
definition struct

```
 {
   int count;
   float *p;
 } *ptr;
```

Then the statement

++ptr → count;

increments count, not ptr.

However ,

(++ptr) → count;

increments ptr first and then links count.

UNIT V STRUCTURES AND UNIONS 9

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.

11.2 STRUCTURE DEFINITION

Unlike arrays, structure must be defined first for their format that may be used later to declare structure variables. Let us use an example to illustrate the process of structure definition and the creation of structure variables. Consider a book database consisting of book name, author, number of pages, and price. We can define a structure to hold this information as follows:

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

The keyword struct declares a structure to hold the details of four data fields, namely title, author, pages, and price. These fields are called structure elements or members. Each member may belong to different type of data. book _ bank is the name of the structure and is called the structure tag. The tag name may be used subsequently to declare variables that have the tag's structure.

The general format of a structure definition is as follows:

```
struct tag _ name
{
    data _ type     member1;
    data _ type     member2;
      ---------      -----
      ---------      -----
};
```

In defining a structure, we may note the following syntax:

1. The template is terminated with a semicolon.
2. While the entire definition is considered as a statement, each member is declared independently for its name and type in a separate statement inside the template.
3. The tag name such as book _ bank can be used to declare structure variables of its type, later in the program.

ARRAY VS STRUCTURE

1. An array is a collection of related data elements of same type. Structure can have elements of different types.
2. An array is derived data type whereas structure is a programmer-defined one.
3. Any array behaves like a built- in data type. All we have to do is to declare an array variable and use it. But in the case of a structure, first we have to design and declare a data structure before the variables of that type are declared and used.

STRUCTURES WITHIN STRUCTURES

Structures within structures means nesting of structures. Nesting of structures is permitted in C. Let us consider the following structure defined to store information about the salary of employees.

```
struct salary
 {
     char name[20];
     char department[10];
     int      basic _ pay;
     int      dearness_ allowance;
     int      house _ rent _ allowance;
     int      city_ allowance;
 }
      employee;
```

This structure defines name, department , basic pay and three kinds of allowances. We can group all items related to allowance together and declare them under a substructure as shown below:

```
struct salary
{
char name[20];
char department[10];
struct
{
int dearness;
int house_rent;
int city;
}
allowance;
}
employee;
```

An inner structure can have more than one variable. The following form of declaration is legal:

```
struct salary
{
……
struct
```

```
int dearness;
…….
}
allowance,
arrears;
}
employee[100];
```

It is also possible to nest more than one type of structures.

```
struct personal_record
{
struct name_part name;
struct addr_part address;
struct date date _ of _
birth ……..
……..
};
struct personal_record person 1;
```

The first member of this structure is name which is of the type struct name_part. Similarly, other members have their structure types.

12.3 STRUCTURES AND FUNCTIONS

C supports the passing of structure values as arguments to functions. There are three methods by which the values of a structure can be transferred from one function to another.

The first method is to pass each member of the structure as an actual argument of the function call.

The second method involves passing of a copy of the entire structure to the called function.

The third approach employs a concept called pointers to pass the structure as an argument.

The general format of sending a copy of a structure to the called function is:

function name(structure variable name)

The called function takes the following form:

```
data_type function name(st_name)
struct_ type st_name;
 {
      …….
      ……..
       return (expression);
 }
```

The following points are important to note:

1. The called function must be declared for its type, appropriate to the data type it is expected to return. For example, if it is returning a copy of the entire structure, then it must be declared as struct with an appropriate tag name.
2. The structure variable used as the actual argument and the corresponding formal argument in the called function must be of the same struct type.
3. The return statement is necessary only when the function is returning some data. The expression may be any simple variable or structure variable or an expression using simple variables.
4. When a function returns a structure, it must be assigned to a structure of identical type in the calling function.
5. The called function must be declared in the calling function for its type, if it is placed after the calling function.

Check Your Progress

Ex 1) Can we nest the structures?

---

2) Can we use arrays within structure?

12.4 UNIONS

Like structures, a union can be declared using the keyword union as follows:

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

This declares a variable code of type union item.

The compiler allocates a piece of storage that is large enough to hold the largest variable type in the union.

To access a union member, we can use the same syntax that we use for structure members. That is,

```
code.m
code.x
code.c
```

are all valid member variables. During accessing, we should make sure that we are accessing the member whose value is currently stored. For example, the statement such as

```
code.m = 379;
code.x=7859.36;
printf("%d", code.m);
```

would produce erroneous output.

In effect, a union creates a storage location that can be used by any one of its members at a time. When a different member is assigned a new value, the new value supercedes the previous member's value.

12.5 SIZE OF STRUCTURES

We normally use structures, unions and arrays to create variables of large sizes. The actual size of these variables in terms of bytes may change from machine to machine. We may use the unary operator sizeof to tell us the size of a structure. The expression

```
sizeof(struct x)
```

will evaluate the number of bytes required to hold all the members of the structure x. If y is a simple structure variable of type struct x, then the expression

```
sizeof(y)
```

would also give the same answer. However, if y is an array variable of type struct x, then

```
sizeof(y)
```

would give the total number of bytes the array requires.

This kind of information would be useful to determine the number of records in a database. For example, the expression

sizeof(y) / sizeof(x)

would give the number of elements in the array y.

12.6 BIT FIELDS

C permits us to use small bit fields to hold data items and thereby to pack several data items in a word of memory. Bit fields allow direct manipulation of string of a string of preselected bits, as if it is represented an integral quantity.

A bit field is a set of adjacent bits whose size can vary from 1 to 16 bits in length. A word can be divided into a number of bit fields. The name and size of bit fields are defined using a structure.

The general form of bit filed definition is

```
struct tag-name
  {
     data-type name1 : bit-length;
     data-type name2 : bit-length;
     data-type name3 : bit-length;
     ---------
     ---------
     ---------
     data-type nameN : bit- length;
  }
```

The data type is either int or unsigned int or signed int and the bit- length is the number of bits used for the specific name. The bit- length is decided by the range of value to be stored.

The largest value that can be stored is 2^{n-1}, where n is bit-length. The internal representation of bit- field is machine dependent. It depends on the size of int and the ordering of bits.

Example :

Suppose we want to store and use the personal information of employees in compressed form. This can be done as follows:

```
struct personal
 {
     unsigned sex:          1
     unsigned age :         7
     unsigned   m_status:   1
     unsigned children:     3
     unsigned           :   4
} emp;
```

This defines a variable name emp with 4 bit fields. The range of values each filed could have is as follows:

Bit Filed	Bit length	Range of values
sex	1	0 or 1
age	7	0 to 127 ($2^7 - 1$)
m_status	1	0 or 1
children	3	0 to 7 ($2^3 - 1$)

The following statements are valid :

```
emp.sex =1 ;
emp.age = 50;
```

It is important to note that we can not use scanf to read the values in to the bit field.

11.4 GIVING VALUES TO MEMBERS

We can access and assign values to the members of a structure in a number of ways. The members themselves are not variables. They should be linked to the structure variables in order to make them meaningful members. For example, the word title has no meaning, whereas the phrase 'title of book' has a meaning. The link between a member and a variable is established using the member operator '.',which is also known as 'dot operator' or 'period operator'. For example,

book1.price

is the variable representing the price of the book1 and can be treated like any other ordinary variable. Here is how we would assign values to the member of book1:

```
strcpy(book1.title, "COBOL");
strcpy(book1.author, "M.K.ROY");
book1.pages = 350;
book1. price =140;
```

We can also use scanf to give the values through the keyboard.

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

are valid input statements.

Example :

Define a structure type, struct personal, that would contain person name, date of joining and salary. Using this structure, write a program to read this information for one person from the keyboard and print the same on the screen.

Structure definition along with the program is shown below. The scanf and printf functions illustrate how the member operator '.' is used to link the structure members to the structure variables. The variable name with a period and the member name is used like an ordinary variable.

Program

```
/*****************************************************************/ /*
DEFINING AND ASSIGNING VALUES TO STRUCTURE MEMBERS */
/*****************************************************************/
struct personal
 {
    char name[20];
    int day;
    char month[10];
    int year;
    float salary;
 };
 main()
  {
    struct personal person;
    printf("Input values\n");
    scanf("%s %d %s %d %f",
                person .name,
               &person. day,
                person.month,
               &person.year,
               &person.salary);
    printf("%s %d %s %d %.2f\n",
                 person .name,
                person. day,
                person.month,
                person.year,
                person.salary);
}
```

11.5 STRUCTURE INITIALIZATION

Like any other data type, a structure variable can be initialized at compile time. main()

```
{
struct
{
int weight;
float height;
}
student ={60, 180.75};
……
……
}
```

This assigns the value 60 to student. weight and 180.75 to student. height. There is a one-to-one correspondence between the members and their initializing values.

A lot of variation is possible in initializing a structure. The following statements initialize two structure variables. Here, it is essential to use a tag name.

```
main()
{
struct st _ record
{
int weight;
float
height; };
struct st_record student1 ={60, 180.75};
struct st_record student2 ={53, 170.60};
……
……
}
```

C language does not permit the initialization of individual structure member within the template. The initialization must be done only in the declaration of the actual variables.

11.6 COMPARISON OF STRUCTURE VARIABLES

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

Operation	Meaning
person1 = person2	Assign person2 to person1.
person1 = =person2	Compare all members of person1 and person2 and return 1 if they are equal, 0 otherwise.
person1 != person2	Return 1 if all the members are not equal, 0 otherwise.

Note that not all compilers support these operations. For example, Microsoft C version does not permit any logical operations on structure variables. In such cases, individual member can be compared using logical operators.

11.7 ARRAYS OF STRUCTURES

We use structure to describe the format of a number of related variables. For example, in analyzing the marks obtained by a class of students, we may use a template to describe student name and marks obtained in various subjects and then declare all the students as structure variables. In such cases, we may declare an array of structure, each elements of the array representing a structure variable. For example,

```
struct class student[100];
```

It defines an array called student, that consists of 100 elements. Each elements is defined to be of the type struct class. Consider the following declaration:

```
struct marks
{
   int subject1;
```

```
Int ubject2; int subject3;
                    };
                    main()
                     {
                       static struct marks student[3] =
                                { {45, 68, 81}, {75, 53, 69}, {57,36,71}};
```

This declares the student as an array of three elements students[0], student[1], and student[2] and initializes their members as follows:

```
student[0].subject1=45;
student[0].subject2=68;
      ……
      ……
student[2].subject3=71;
```

An array of structures is stored inside the memory in the same way as a multi-dimensional array.

Check Your Progress

Ex 1) Can we compare structure variables ? If so, how?

Ex 2) Construct a structure for bank details.

Student[0].subject1	45
.subject2	68
.subject3	81
Student[1].subject1	75
.subject2	53
.subject3	69
Student[2].subject1	

57

.subject2

36

.subject3

71

The array student inside memory.

STORAGE CLASSES

AUTOMATIC VARIABLES (LOCAL/INTERNAL)

Automatic variables are declared inside a function in which they are to be utilized.

They are created when a function is called and destroyed automatically when the function is exited.

```
Eg:main()
{
int number;
---------
---------
}
```

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

EXTERNAL VARIABLES

Variables that are both alive and active throughout the entire program are known as external variables. They are also known as global variables.

```
Eg: int
numb
er;
float
length
= 7.5;
main()
{
-------
-------
}
function1( )
{
-------
-------
```

```
}
function2( )
{
-------
-------
}
```

The keyword **extern** can be used for explicit declarations of external variables.

STATIC VARIABLES

As the name suggests, the value of a static variable persists until the end of the program. A variable can be declared static using the keyword **static.**

Eg:1) **static int x;**
2) static int y;

REGISTER VARIABLES

We can tell the compiler that a variable should be kept in one of the machine's registers, instead of keeping in the memory. Since a register access is much faster than a memory access, keeping the frequently accessed variables in the register will lead to faster execution of programs. This is done as follows:

register int count;

Check Your Progress

Ex 1) Is the keyword auto is compulsory in declaration?

---

2) What is the other name for external variables?

---

3) Can we declare all variables as register variables?

ANSI C FUNCTIONS

The general form of ANSI C function is

```
data-type function-name(type1 a1,type2 a2,…..typeN aN)
{
  --------
  --------     (body of the function)
  --------
}
```

Eg: 1) **double funct(int a, int b, double c)**

Function Declaration

The general form of function declaration is

data-type **function-name**(type1 a1,type2 a2,…..typeN aN)

```
Eg:main()
{
float a, b, x;
float mul(float length,float breadth);/*declaration*/
---------
---------
x = mul(a,b);
}
```

THE PREPROCESSOR

The Preprocessor, as the name implies, is a program that processes the source code before it passes through the complier. It operates under the control of preprocessor command lines or directives. Preprocessor directives are placed in the source program before the main line. Before the source code passes through the compiler, it is examined by the preprocessor for any preprocessor directives. If there are any, appropriate actions (as per the directives) are taken and then the source program is handed over to the compiler.

Preprocessor directives begin with the symbol # in column one and do not require a semicolon at the end.

Commonly used Preprocessor directives

Directive	**Function**

#define	Defines a macro substitution
#undef	Undefines a macro
#include	specifies the files to be include
# ifdef	Tests for a macro definition
#endif	specifies the end of #if
#ifndef	Tests whether a macro is not defined
#if	Tests a compile time condition
#else	specifies alternatives when #if fails

Preprocessor directives can be divided into three categories

1) Macro substitution directives
2) File Inclusion directives
3) Compiler control directives

Macro Substitution directive

The general form is

#define identifier string

Examples 1) #define COUNT 100 (simple macro
2) #define CUBE(x) x*x*x (macro with arguments)
3) # define M 5
#define N M+1 (nesting of macros)

File Inclusion directive

This is achieved by

#include "filename" or #include <filename>

Examples 1) #include
<stdio.h> 2)
#include
“TEST.C”

Check Your Progress

Ex 1) Give examples for macro substitution directives

Ex 2) Differentiate #include <…> and #include “…”

Compiler control directives

These are the directives meant for controlling the compiler actions. C preprocessor offers a feature known as conditional compilation, which can be used to switch off or on a particular line or group of lines in a program. Mostly #ifdef and #ifndef are used in these directives.