SAMARTH (PRIVATE) INDUSTRIAL TRAINING INSTITUTE, BELHE.

Trade : Computer Operator & Progr amming Assistant **LESSION PLAN**

Semester Code: COP: SEM I (Aug 2013 TO Feb 2014) **(Subject: Theory)**

The History of Computers

Week No: 01

"Who invented the computer?" is not a question with a simple answer. The real answer is that many inventors contributed to the history of computers and that a computer is a complex piece of machinery made up of many parts, each of which can be considered a separate invention.

This series covers many of the major milestones in computer history (but not all of them) with a concentration on the history of personal home computers.

Computer History Year/Enter	**Computer History Inventors/Inventions**	**Computer History Description of Event**
1936	Konrad Zuse - **Z1 Computer**	First freely programmable computer.
1942	John Atanasoff & Clifford Berry **ABC Computer**	Who was first in the computing biz is not always as easy as ABC.
1944	Howard Aiken & Grace Hopper **Harvard Mark I Computer**	The Harvard Mark 1 computer.
1946	John Presper Eckert & John W. Mauchly **ENIAC 1 Computer**	20,000 vacuum tubes later...
1948	Frederic Williams & Tom Kilburn **Manchester Baby Computer & The Williams Tube**	Baby and the Williams Tube turn on the memories.

1947/48	John Bardeen, Walter Brattain & William Shockley **The Transistor**	No, a transistor is not a computer, but this invention greatly affected the history of computers.
1951	John Presper Eckert & John W. Mauchly **UNIVAC Computer**	First commercial computer & able to pick presidential winners.
1953	International Business Machines **IBM 701 EDPM Computer**	IBM enters into 'The History of Computers'.
1954	John Backus & IBM **FORTRAN Computer Programming Language**	The first successful high level programming language.
1955 (In Use 1959)	Stanford Research Institute, Bank of America, and General Electric **ERMA and MICR**	The first bank industry computer - also MICR (magnetic ink character recognition) for reading checks.
1958	Jack Kilby & Robert Noyce **The Integrated Circuit**	Otherwise known as 'The Chip'
1962	Steve Russell & MIT **Space war Computer Game**	The first computer game invented.
1964	Douglas Engelbart **Computer Mouse & Windows**	Nicknamed the mouse because the tail came out the end.
1969	**Arpanet**	The original Internet.
1970	**Intel 1103 Computer Memory**	The world's first available dynamic RAM chip.
1971	Fagin, Hoff & Mazor **Intel 4004 Computer Microprocessor**	The first microprocessor.

1971	Alan Shugart &IBM **The "Floppy" Disk**	Nicknamed the "Floppy" for its flexibility.
1973	Robert Metcalfe & Xerox **The Ethernet Computer Networking**	Networking.
1974/75	**Scelbi & Mark-8 Altair & IBM 5100 Computers**	The first consumer computers.
1976/77	**Apple I, II & TRS-80 & Commodore Pet Computers**	More first consumer computers.
1978	Dan Bricklin & Bob Frankston **VisiCalc Spreadsheet Software**	Any product that pays for itself in two weeks is a surefire winner.
1979	Seymour Rubenstein & Rob Barnaby **WordStar Software**	Word Processors.
1981	IBM **The IBM PC - Home Computer**	From an "Acorn" grows a personal computer revolution
1981	Microsoft **MS-DOS Computer Operating System**	From "Quick And Dirty" comes the operating system of the century.
1983	**Apple Lisa Computer**	The first home computer with a GUI, graphical user interface.
1984	**Apple Macintosh Computer**	The more affordable home computer with a GUI.
1985	**Microsoft Windows**	Microsoft begins the friendly war with Apple.
SERIES	TO BE	CONTINUED

A TYPICAL COMPUTER SYSTEM

A typical digital computer consists of:

a) A central processor unit (CPU)
b) A memory
c) Input/output (1/0) ports

The memory serves as a place to store Instructions, the coded pieces of information that direct the activities of the CPU, and Data, the coded pieces of information that are processed by the CPU. A group of logically related instructions stored in memory is referred to as a Program. The CPU "reads" each instruction from memory in a logically determined sequence, and uses it to initiate processing actions. If the program sequence is coherent and logical, processing the program will produce intelligible and useful results.

The memory is also used to store the data to be manipulated, as well as the instructions that direct that manipulation The program must be organized such that the CPU does not read a non-instruction word when it expects to see an instruction. The CPU can rapidly access any data stored in memory; but often the memory is not large enough to store the entire data bank required for a particular application. The problem can be resolved by providing the computer with one or more Input Ports. The CPU can address these ports and input the data contained there. The addition of input ports enables the computer to receive information from external equipment (such as a paper tape reader or floppy disk) at high rates of speed and in large volumes.

A computer also requires one or more Output Ports that permit the CPU to communicate the result of its processing to the outside world. The output may go to a display, for use by a human operator, to a peripheral device that produces "hardcopy," such as a line-printer, to a peripheral storage device, such as a floppy disk unit, or the output may constitute process control signals that direct the operations of another system, such as an automated assembly line. Like input ports, output ports are addressable. The input and output ports together permit the processor to communicate with the outside world.

The CPU unifies the system. It controls the functions performed by the other components. The CPU must be able to fetch instructions from memory, decode their binary contents and execute them. It must also be able to reference memory and 1/0 ports as necessary in the execution of instructions. In addition, the CPU should be able to recognize and respond to certain external control signals, such as INTERRUPT and WAIT requests. The functional units within a CPU that enable it to perform these functions are described below.

THE ARCHITECTURE OF A CPU

A typical central processor unit (CPU) consists of the following interconnected functional units:

Registers

Arithmetic/Logic Unit (ALU)

Control Circuitry

Registers are temporary storage units within the CPU. Some registers, such as the program counter and instruction register, have dedicated uses. Other registers, such as the accumulator, are for more general purpose use.

Accumulator:

The accumulator usually stores one of the operands to be manipulated by the ALU. A typical instruction might direct the ALU to add the contents of some other register to the contents of the accumulator and store the result in the accumulator itself. In general, the accumulator is both a source (operand) and a destination (result) register.

Often a CPU will include a number of additional general purpose registers that can be used to store operands or intermediate data. The availability of general purpose registers eliminates the need to "shuffle" intermediate results back and forth between memory and the accumulator, thus improving processing speed and efficiency.

Program Counter (Jumps, Subroutines and the Stack):

The instructions that make up a program are stored in the system's memory. The central processor references the contents of memory, in order to determine what action is appropriate. This means that the processor must know which location contains the next instruction.

Each of the locations in memory is numbered, to distinguish it from all other locations in memory. The number which identifies a memory location is called its Address.

The processor maintains a counter which contains the address of the next program instruction. This register is called the Program Counter. The processor updates the program counter by adding "1" to the counter each time it fetches an instruction, so that the program counter is always current (pointing to the next instruction)

The programmer therefore stores his instructions in numerically adjacent addresses, so that the lower addresses contain the first instructions to be executed and the higher addresses contain later instructions. The only time the programmer may violate this sequential rule is when an instruction in one section of memory is a Jump instruction to another section of memory.

A jump instruction contains the address of the instruction which is to follow it. The next instruction may be stored in any memory location, as long as the programmed jump specifies the correct address. During the execution of a jump instruction, the processor replaces the contents of its program counter with the address embodied in the Jump. Thus, the logical continuity of the program is maintained.

A special kind of program jump occurs when the stored program "Calls" a subroutine. In this kind of jump, the processor is required to "remember" the contents of the program counter at the time that the jump occurs. This enables the processor to resume execution of the main program when it is finished with the last instruction of the subroutine.

A Subroutine is a program within a program. Usually it is a general-purpose set of instructions that must be executed repeatedly in the course of a main program. Routines which calculate the square, the sine, or the logarithm of a program variable are good examples of functions often written as subroutines. Other examples might be programs designed for inputting or outputting data to a particular peripheral device.

The processor has a special way of handling subroutines, in order to insure an orderly return to the main program. When the processor receives a Call instruction, it increments the Program Counter and stores the counter's contents in a reserved memory area known as the Stack. The Stack thus saves the address of the instruction to be executed after the subroutine is completed. Then the processor loads the address specified in the Call into its Program Counter. The next instruction fetched will therefore be the first step of the subroutine.

The last instruction in any subroutine is a Return. Such an instruction need specify no address. When the processor fetches a Return instruction, it simply replaces the current contents of the Program Counter with the address on the top of the stack. This causes the processor to resume execution of the calling program at the point immediately following the original Call Instruction.

Subroutines are often Nested; that is, one subroutine will sometimes call a second subroutine. The second may call a third, and so on. This is perfectly acceptable, as long as the processor has enough capacity to store the necessary

return addresses, and the logical provision for doing so. In other words, the maximum depth of nesting is determined by the depth of the stack itself. If the stack has space for storing three return addresses, then three levels of subroutines may be accommodated.

Processors have different ways of maintaining stacks. Some have facilities for the storage of return addresses built into the processor itself. Other processors use a reserved area of external memory as the stack and simply maintain a Pointer register which contains the address of the most recent stack entry. The external stack allows virtually unlimited subroutine nesting. In addition, if the processor provides instructions that cause the contents of the accumulator and other general purpose registers to be "pushed" onto the stack or "popped" off the stack via the address stored in the stack pointer, multilevel interrupt processing (described later in this chapter) is possible. The status of the processor (i.e., the contents of all the registers) can be saved in the stack when an interrupt is accepted and then restored after the interrupt has been serviced. This ability to save the processor's status at any given time is possible even if an interrupt service routine, itself, is interrupted.

Instruction Register and Decoder:

Every computer has a Word Length that is characteristic of that machine. A computer's word length is usually determined by the size of its internal storage elements and interconnecting paths (referred to as Busses); for example, a computer whose registers and busses can store and transfer 8 bits of information has a characteristic word length of 8 bits and is referred to as an 8bit parallel processor. An eight-bit parallel processor generally finds it most efficient to deal with eight-bit binary fields, and the memory associated with such a processor is therefore organized to store eight bits in each addressable memory location. Data and instructions are stored in memory as eight-bit binary numbers, or as numbers that are integral multiples of eight bits: 16 bits, 24 bits, and so on. This characteristic eight-bit field is often referred to as a Byte.

Each operation that the processor can perform is identified by a unique byte of data known as an Instruction Code or Operation Code. An eight-bit word used as an instruction code can distinguish between 256 alternative actions, more than adequate for most processors.

The processor fetches an instruction in two distinct operations. First, the processor transmits the address in its Program Counter to the memory Then the memory returns the addressed byte to the processor. The CPU stores this instruction byte in a register known as the Instruction Register, and uses it to direct activities during the remainder of the instruction execution.

The mechanism by which the processor translates an instruction code into specific processing actions requires more elaboration than we can here afford. The concept, however, should be intuitively clear to any logic designer.

The eight bits stored in the instruction register can be decoded and used to selectively activate one of a number of output lines, in this case up to 256 lines. Each line represents a set of activities associated with execution of a particular instruction code. The enabled line can be combined with selected timing pulses, to develop electrical signals that can then be used to initiate specific actions. This translation of code into action is performed by the Instruction Decoder and by the associated control circuitry.

An eight-bit instruction code is often sufficient to specify a particular processing action. There are times, however, when execution of the instruction requires more information than eight bits can convey

One example of this is when the instruction references a memory location. The basic instruction code identifies the operation to be performed, but cannot specify the object address as well In a case like this, a two or three-byte instruction must be used. Successive instruction bytes are stored in sequentially adjacent memory locations, and the processor performs two or three fetches in succession to obtain the full instruction. The first byte retrieved from memory is placed in the processor's instruction register, and subsequent bytes are placed in temporary storage; the processor then proceeds with the execution phase. Such an instruction is referred to as Variable Length.

Address Register(s):

A CPU may use a register or register pair to hold the address of a memory location that is to be accessed for data If the address register is Programmable, (i e., if there are instructions that allow the programmer to alter the contents of the register) the program can "build" an address in the address register prior to executing a Memory Reference instruction (i.e., an instruction that reads data from memory, writes data to memory or operates on data stored in memory).

Arithmetic/Logic Unit (ALU):

All processors contain an arithmetic/logic unit, which is often referred to simply as the ALU The ALU, as its name implies, is that portion of the CPU hardware which performs the arithmetic and logical operations on the binary data .

The ALU must contain an Adder which is capable of combining the contents of two registers in accordance with the logic of binary arithmetic. This provision permits the processor to perform arithmetic manipulations on the data it obtains from memory and from its other inputs.

Using only the basic adder a capable programmer can write routines which will subtract, multiply and divide, giving the machine complete arithmetic capabilities. In practice, however, most ALUs provide other built-in functions, including hardware subtraction, Boolean logic operations, and shift capabilities

The ALU contains Flag Bits which specify certain conditions that arise in the course of arithmetic and logical manipulations. Flags typically include Carry, Zero, Sign, and Parity. It is possible to program jumps which are conditionally dependent on the status of one or more flags. Thus, for example, the program may be designed to jump to a special routine if the carry bit is set following an addition instruction

Control Circuitry:

The control circuitry is the primary functional unit within a CPU. Using clock inputs, the control circuitry maintains the proper sequence of events required for any processing task After an instruction is fetched and decoded, the control circuitry issues the appropriate signals (to units both internal and external to the CPU) for initiating the proper processing action. Often the control circuitry will be capable of responding to external signals, such as an interrupt or wait request An Interrupt request will cause the control circuitry to temporarily interrupt main program execution, jump to a special routine to service the interrupting device, then automatically return to the main program. A Wait request is often issued by a memory or 1/0 element that operates slower than the CPU. The control circuitry will idle the CPU until the memory or 1/0 port is ready with the data.

COMPUTER OPERATIONS

There are certain operations that are basic to almost any computer A sound understanding of these basic operations is a necessary prerequisite to examining the specific operations of a particular computer.

Timing:

The activities of the central processor are cyclical. The processor fetches an instruction, performs the operations
required, fetches the next instruction, and so on. This orderly sequence of events requires precise timing, and the CPU therefore requires a free running oscillator clock which furnishes the reference for all processor actions The combined fetch and execution of a single instruction is referred to as an Instruction Cycle. The portion of a cycle identified with a clearly defined activity IS called a State. And the inter vat between pulses of the timing oscillator is referred to as a Clock Period. As a general rule, one or more clock periods are necessary for the completion of a state, and there are several states in a cycle.

Instruction Fetch:

The first state(s) of any instruction cycle will be dedicated to fetching the next instruction. The CPU issues a read signal and the contents of the program counter are sent to memory, which responds by returning the next instruction word. The first byte of the instruction is placed in the instruction register. If the instruction consists of more than one byte, additional states are required to fetch each byte of the instruction. When the entire instruction is present in the CPU, the program counter is incremented (in preparation for the next instruction fetch) and the instruction is decoded. The operation specified in the instruction will be executed in the remaining states of the instruction cycle. The instruction may call for a memory read or write, an input or output and/or an internal CPU operation, such as a register to register transfer or an add registers operation.

Memory Read:

An instruction fetch is merely a special memory read operation that brings the instruction to the CPU's instruction register. The instruction fetched may then call for data to be read from memory into the CPU. The CPU again issues a read signal and sends the proper memory address; memory responds by returning the requested word. The data received is placed in the accumulator or one of the other general purpose registers (not the instruction register).

Memory Write:

A memory write operation is similar to a read except for the direction of data flow. The CPU issues a write signal, sends the proper memory address, then sends the data word to be written into the addressed memory location.

Wait (memory synchronization):

As previously stated, the activities of the processor are timed by a master clock oscillator. The clock period determines the timing of all processing activity.

The speed of the processing cycle, however, is limited by the memory's Access Time. Once the processor has sent a read address to memory, it cannot proceed until the memory has had time to respond. Most memories are capable of responding much faster than the processing cycle requires. A few, however, cannot supply the addressed byte within the minimum time established by the processor's clock.

Therefore a processor should contain a synchronization provision, which permits the memory to request a Wait state. When the memory receives a read or write enable signal, it places a request signal on the processor's READY line, causing the CPU to idle temporarily. After the memory has had time to respond, it frees the processor's READY line, and the instruction cycle proceeds

Input/output:

Input and Output operations are similar to memory read and write operations with the exception that a peripheral 1/0 device is addressed instead of a memory location. The CPU issues the appropriate input or output control signal, sends the proper device address and either receives the data being input or sends the data to be output.
Data can be input/output in either parallel or serial form. All data within a digital computer is represented in binary coded form. A binary data word consists of a group 5 of bits; each bit is either a one or a zero. Parallel 1/0 consists of transferring all bits in the word at the same time, one bit per line. Serial 1/0 consists of transferring one bit at a

time on a single line. Naturally serial 1/0 is much slower, but it requires considerably less hardware than does parallel 1/0.

Interrupts:

Interrupt provisions are included on many central processors, as a means of improving the processor's efficiency. Consider the case of a computer that is processing a large volume of data, portions of which are to be output to a printer. The CPU can output a byte of data within a single machine cycle but it may take the printer the equivalent of many machine cycles to actually print the character specified by the data byte. The CPU could then remain idle waiting until the printer can accept the next data byte. If an interrupt capability is implemented on the computer, the CPU can output a data byte then return to data processing. When the printer is ready to accept the next data byte, it can request an interrupt. When the CPU acknowledges the interrupt, it suspends main program execution and automatically branches to a routine that will output the next data byte. After the byte is output, the CPU continues with main program execution. Note that this is, in principle, quite similar to a subroutine call, except that the jump is initiated externally rather than by the program.

More complex interrupt structures are possible, in which several interrupting devices share the same processor but have different priority levels. Interruptive processing is an important feature that enables maximum utilization of a processor's capacity for high system throughput.

Hold:

Another important feature that improves the throughput of a processor is the Hold. The hold provision enables Direct Memory Access (DMA) operations.

In ordinary input and output operations, the processor itself supervises the entire data transfer. Information to be placed in memory is transferred from the input device to the processor, and then from the processor to the designated memory location. In similar fashion, information that goes from memory to output devices goes by way of the processor.

Some peripheral devices, however, are capable of transferring information to and from memory much faster than the processor itself can accomplish the transfer. If any appreciable quantity of data must be transferred to or from such a device, then system throughput will be increased by having the device accomplish the transfer directly. The processor must temporarily suspend its operation during such a transfer, to prevent conflicts that would arise if processor and peripheral device attempted to access memory simultaneously. It is for this reason that a hold provision is included on some processors.

COMPUTER HARDWARE Week: 04

Contents

Personal computer hardware is the component **devices** that are the building blocks of personal computers. These are typically installed into a computer case, or attached to it by acable or through a port. In the latter case, they are also referred to as peripherals.

Case

A computer case (also known as a box) is a box that has bits of computer in it (usually excluding the display, keyboard and mouse). A computer case is sometimes referred to eponymously as a DMA meaning Dome Media Aphonics. DMA was a common term in the earlier days of home computers, when conjunctions other than the father board were usually housed in their own separate cases.

Power supply

A power supply unit (PSU) converts alternating current (AC) electric power to low-voltage DC power for the internal components of the computer. Some power supplies have a switch to change between 230 V and 115 V. Other models have automatic sensors that switch input voltage automatically, or are able to accept any voltage between those limits. Power supply units used in computers are nearly always switch mode power supplies (SMPS). The SMPS provides regulated direct current power at the several voltages required by the motherboard and accessories such as disk drives and cooling fans.

Motherboard

The motherboard is the main component inside the case. It is a large rectangular board with integrated circuitry that connects the other parts of the computer including the CPU, the RAM, the disk drives (CD, DVD, hard disk, or any others) as well as any peripherals connected via the ports or the expansion slots.

Components directly attached to the motherboard include:

The **CPU** (Central Processing Unit) performs most of the calculations which enable a computer to function, and is sometimes referred to as the "brain" of the computer. It is usually cooled by a heat sink and fan. Most newer CPUs include an on-die Graphics Processing Unit (GPU).

The **Chipset**, which includes the north bridge, mediates communication between the CPU and the other components of the system, including main memory.

The **Random-Access Memory** (RAM) stores the code and data that are being actively accessed by the CPU.

The **Read-Only Memory** (ROM) stores the BIOS that runs when the computer is powered on or otherwise begins execution, a process known as Bootstrapping, or "booting" or "booting up". The **BIOS** (Basic Input Output System) includes boot firmware and power management firmware. Newer motherboards use Unified Extensible Firmware Interface (UEFI) instead of BIOS.

Buses connect the CPU to various internal components and to expansion cards for graphics and sound.

Current

PCI Express: for expansion cards such as graphics, sound, network interfaces, TV tuners, etc.

PCI: for other expansion cards.

SATA: for disk drives.

Obsolete

AGP: superseded by PCI Express.

ATA

VLB: VESA Local Bus, superseded by AGP.

EISA

Micro Channel architecture

ISA: expansion card slot format obsolete in PCs, but still used in industrial computers.

Ports for external peripherals. These ports may be controlled directly by the south bridge I/O controller or provided by expansion cards attached to the motherboard.

USB

Memory Card

FireWire

eSATA

SCSI

Expansion cards

The expansion card (also expansion board, adapter card or accessory card) in computing is a printed circuit board that can be inserted into an expansion slot of a computer motherboard or backplane to add functionality to a computer system via the expansion bus.

An example of an expansion card is a sound card that enables the computer to output sound to audio devices, as well as accept input from a microphone. Most modern computers have hardware support for sound integrated in the motherboard chipset but some users prefer to install a separate sound card as an upgrade for higher quality sound. Most sound cards, either built-in or added, have surround sound capabilities and 3-D sound effects.

Secondary storage devices

Computer data storage, often called storage or memory, refers to computer components and recording media that retain digital data. Data storage is a core function and fundamental component of computers.

Fixed media

Hard disk drives: a hard disk drive (HDD; also hard drive, hard disk, or disk drive)[2] is a device for storing and retrieving digital information, primarily computer data. It consists of one or more rigid (hence "hard") rapidly rotating discs (often referred to as platters), coated with magnetic material and with magnetic heads arranged to write data to the surfaces and read it from them.

Solid-state drives: a solid-state drive (SSD), sometimes called a solid-state disk or electronic disk, is a data storage device that uses solid-state memory to store persistent data with the intention of providing access in the same manner of a traditional block I/O hard disk drive. SSDs are distinguished from traditional magnetic disks such as hard disk drives (HDDs) or floppy disk, which are electromechanical devices containing spinning disks and movable read/write heads.

RAID array controller - a device to manage several internal or external hard disks and optionally some peripherals in order to achieve performance or reliability improvement in what is called a RAID array.

Removable media

Optical Disc Drives for reading from and writing to various kinds of optical media, including Compact Discs such as ROMs, DVDs, DVD-RAMs and Blu-ray Discs. Optical discs are the most common way of transferring digital video, and are popular for data storage as well.

Floppy disk drives for reading and writing to floppy disks, an outdated storage media consisting of a thin disk of a flexible magnetic storage medium. These were once standard on most computers but are no longer in common use. Floppies are used today mainly for loading device drivers not included with an operating system release (for example, RAID drivers).

Zip drives, an outdated medium-capacity removable disk storage system, for reading from and writing to Zip disks, was first introduced by Iomega in 1994.

USB flash drive plug into a USB port and do not require a separate drive. USB flash drive is a typically small, lightweight, removable, and rewritable flash memory data storage device integrated with a USB interface. Capacities vary, from hundreds of megabytes (in the same range as CDs) to tens of gigabytes (surpassing Blu-ray discs but also costing significantly more).

Memory card readers for reading from and writing to Memory cards, a flash memory data storage device used to store digital information. Memory cards are typically used on mobile devices. They are thinner, smaller and lighter than USB flash drives. Common types of memory cards are SD and MS.

Tape drives read and write data on a magnetic tape, and are used for long term storage and backups.

Input and output peripherals

Input and output devices are typically housed externally to the main computer chassis. The following are either standard or very common to many computer systems.

Input

Input devices allow the user to enter information into the system, or control its operation. Very early computer systems had literal toggle switches that could be tested by running programs as a simple form of user input; modern personal computers have alphanumeric keyboards and pointing devices to allow the user to interact with running software.

Text input devices

Keyboard - a device to input text and characters by depressing buttons (referred to as keys or buttons).

Pointing devices

Mouse - a pointing device that detects two dimensional motion relative to its supporting surface.

Optical Mouse - uses light (laser technology) to determine mouse motion.

Trackball - a pointing device consisting of an exposed protruding ball housed in a socket that detects rotation about two axes.

Touch screen - senses the user pressing directly on the monitor.

Gaming devices

Joystick - a hand-operated pivoted stick whose position is transmitted to the computer.

Game pad - a hand held game controller that relies on the digits (especially thumbs) to provide input.

Game controller - a specific type of controller specialized for certain gaming purposes.

Image, Video input devices

Image scanner - a device that provides input by analyzing images, printed text, handwriting, or an object.

Web cam - a video camera used to provide visual input that can be easily transferred over the internet.

Audio input devices

Microphone - an acoustic sensor that provides input by converting sound into electrical signals.

Output

Output devices display information in a human readable form. A program-controlled pilot lamp would be a very simple example of an output devices. Modern personal computers have full-screen point-addressable graphic displays and often a printing device to produce paper copies of documents and images.

Printer - a device that produces a permanent human-readable text or graphic document.

Laser printer

Inkjet printer

Dot matrix printer

Thermal printer

Computer monitors

Speakers

Source Data input Week no: 05

Definition of 'Point-Of-Sale Terminal'

A type of electronic-transaction terminal. Point-of-sale terminals typically include a computer, a cash register and other equipment or software used to sell goods or services. They also transmit sales data to be posted to customer accounts.

Investopedia explains 'Point-Of-Sale Terminal'

The most common type of terminal is the electronic cash register used by volume-transaction merchants such as department stores. Common among all point-of-sale terminals is the emphasis on speed of operation and ease of use of the hardware and software.

Laser beam scanner

Most laser scanners use moveable mirrors to steer the laser beam. The steering of the beam can be **one-dimensional**, as inside a laser printer, or **two-dimensional**, as in a laser show system.

Additionally, the mirrors can lead to a **periodic** motion - like the rotating mirror polygons in a barcode scanner or so-called resonant galvanometer scanners - or to an **freely addressable** motion, as in servo-controlled galvanometer scanners. One also uses the terms **raster scanning** and **vector scanning** to distinguish the two situations.

To control the scanning motion, scanners need a rotary encoder and control electronics that provides, for a desired angle or phase, the suitable electrical current to the motor or galvanometer. A software systems usually controls the scanning motion and, if 3D scanning is implemented, also the collection of the measured data.

In order to position a laser beam in **two dimensions**, it is possible either to rotate one mirror along two axes - used mainly for slow scanning systems - or to reflect the laser beam onto two closely spaced mirrors that are mounted on orthogonal axes. Each of the two flat or polygonal mirrors is then driven by a galvanometer or by an electric motor. Two-dimensional systems are essential for most applications in material processing, co focal microscopy, and medical science.

Some applications require positioning the focus of a laser beam in **three dimensions**. This is achieved by a servo-controlled lens system, usually called a 'focus shifter' or 'z-shifter'.

Many laser scanners further allow changing the laser intensity.

In laser projectors for laser TV or laser displays, the three fundamental colors red blue and green are combined in a single beam and then reflected together over the two mirrors.

The most common way to move mirrors is, as mentioned, the use of an electric motor or of a galvanometer. However piezoelectric actuators or magnetostrictive actuators are alternative options. They offer higher achievable angular speeds, but often at the expense of smaller achievable maximum angles.

Optical sense reader

Electrographic is a term used for punched card and page scanning technology that allowed cards or pages marked with a pencil to be processed or converted into punched cards. That technology was sold by IBM, its developer, under the term mark sense. A "mark sense pencil lead" sold by IBM would meet federal specifications for "electrographic lead."

Mark sense was a trade name used by IBM for electrographic forms and systems. It has since come to be used as a generic term for any technology allowing marks made using ordinary writing implements to be processed, encompassing both optical mark recognition and electrographic technology, because the user of a mark-sense form cannot generally tell if the marks are sensed electrically or optically. The term **mark sense** is not generally used when referring to technology that distinguishes the shape of the mark; the general term optical character recognition is generally used when mark shapes are distinguished. Because the term mark-sense was originally a trade name, the Federal Government generally used the term electrographic.

In the 1940s, 50s, and 60s, mark sense technology was widely used for applications like recording meter readings and recording long distance telephone calls[Many thousands of pencils were made expressly for mark sense applications by the Dur-O-Lite Pencil Company and by the Auto point Company. Many of the pencils made for the "Bell System" were stamped "MARK SENSE LEAD" and for the Federal Government, "US Government Electrographic."

Reynolds Johnson was a teacher who set out to develop an automatic test scoring machine. The result of this work was the IBM 805 Test Scoring Machine. IBM hired Johnson as an engineer, and he went on to develop a range of electrographic mark-sense machinery. The first large-scale use of the IBM 805 was by the American Council on Education's Cooperative Test Service in 1936; In 1947, the Cooperative Test Service became part of the Educational Testing Service.

Various IBM equipment could be used with mark sense cards including the IBM 513 and IBM 514 Reproducing Punches, the IBM 557Alphabetic Interpreter, and the IBM 519 Electric Document Originating Machine.

Optical character recognition

Optical character recognition, usually abbreviated to **OCR**, is the mechanical or electronic conversion of scanned images of handwritten, typewritten or printed text into machine-encoded text. It is widely used as a form of data entry from some sort of original paper data source, whether documents, sales receipts, mail, or any number of printed records. It is a common method of digitizing printed texts so that they can be electronically searched, stored more compactly, displayed on-line, and used in machine processes such as machine translation, text-to-speech and text mining. OCR is a field of research in pattern recognition, artificial intelligence andcomputer vision.

Early versions needed to be programmed with images of each character, and worked on one font at a time. "Intelligent" systems with a high degree of recognition accuracy for most fonts are now common. Some systems are capable of reproducing formatted output that closely approximates the original scanned page including images, columns and other non-textual components.

Magnetic ink character recognition

Magnetic ink character recognition, or **MICR**, is a character recognition technology used primarily by the banking industry to facilitate the processing and clearance of cheques and other documents. The MICR encoding, called the MICR line, is located at the bottom of a cheque or other voucher and typically includes the document type indicator, bank code, bank account number, cheque number and the amount, plus some control indicator. The technology allows MICR readers to scan and read the information directly into a data collection device. Unlike barcodes or similar technologies, MICR characters can be easily read by humans. The MICR E-13B font has been adopted as the international standard in ISO 1004:1995, but the CMC-7 font is widely used in Europe.

Output Device

Dot matrix printing

Dot matrix printing or **impact matrix printing** is a type of computer printing which uses a print head that runs back and forth, or in an up and down motion, on the page and prints by impact, striking an ink-soaked cloth ribbon against the paper, much like the print mechanism on a typewriter. However, 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.

Inkjet printing

Inkjet printing is a type of computer printing that creates a digital image by propelling droplets of ink onto paper, plastic, or other substrates. Inkjet printers are the most commonly used type of printer,[1] and range from small inexpensive consumer models to very large professional machines that can cost tens of thousands of dollars, or more.

The concept of inkjet printing originated in the 19th century, and the technology was first extensively developed in the early 1950s. Starting in the late 1970s inkjet printers that could reproduce digital images generated by computers were developed, mainly by Epson, Hewlett-Packard (HP), and Canon. In the worldwide consumer market, four manufacturers account for the majority of inkjet printer sales: Canon, HP, Epson, and Lexmark, a 1991 spin-off from IBM.

The emerging ink jet material deposition market also uses inkjet technologies, typically print heads using piezoelectric crystals, to deposit materials directly on substrates.

There are two main technologies in use in contemporary inkjet printers: continuous (CIJ) and Drop-on-demand (DOD).

Laser printing

Laser printing is an electrostatic digital printing process that rapidly produces high quality text and graphics by passing a laser beam over a charged drum to define a differentially charged image. The drum then selectively collects charged toner and transfers the image to paper, which is then heated to permanently fix the image. As with digital photocopiers and multifunction printers (MFPs), laser printers employ a xerographic printing process, but differ from analog photocopiers in that the image is produced by the direct scanning of the medium across the printer's photoreceptor. Hence, it proves to be a much faster process compared to the latter.

Plotter

The **plotter** is a computer printer for printing vector graphics. In the past, plotters were used in applications such as computer-aided design, though they have generally been replaced with wide-format conventional printers.

Pen plotters print by moving a pen or other instrument across the surface of a piece of paper. This means that plotters are vector graphics devices, rather than raster graphics as with other printers. Pen plotters can draw complex line art, including text, but do so slowly because of the mechanical movement of the pens. They are often incapable of efficiently creating a solid region of color, but can hatch an area by drawing a number of close, regular lines.

Plotters offered the fastest way to efficiently produce very large drawings or color high-resolution vector-based artwork when computer memory was very expensive and processor power was very limited, and other types of printers had limited graphic output capabilities.

Pen plotters have essentially become obsolete, and have been replaced by large-format inkjet printers and LED toner based printers. Such devices may still understand vector languages originally designed for plotter use, because in many uses, they offer a more efficient alternative to raster data.

OUTPUT Devices Week No: 06/07

VDU (visual display unit)

All computers are connected to some type of display unit, which is called a monitor. The monitor is a part of the computer video system and monitors are available in many different types and size. The on-screen display enables us to see how applications are processing our data, but Visual display unit is important to remember that the screen display is not permanent record i.e. the outputs are lost where the power is off.

The thinner monitors used on notebook and other small computer are known as flat-panel displays. Compared to CRT (Cathode Ray Tube) based monitors, flat panel displays consume less electricity and take up much less room. Most flat panel displays use Liquid Crystal Display (LCD) technology.LCD displays sandwich cells containing tiny crystals between two transparent surfaces. By varying the electrical current supplied to each crystal, an images forms.

Factors affecting monitor quality

Quality of monitor is often judged in terms of following four factors:

Monitor size

Dot pitch

Resolution

Refresh rate

Monitor size: The most important aspect of a monitories its size. Like televisions, screen sizes are measured in diagonal inches, the distance from lower left corner to the upper right corner diagonally. Typical monitors found in the market are of size 14 inches, and 17 inches and above. The size of display are determines monitor quality. In larger monitors the objects look bigger or more objects can be fitted on the screen.

Dot pitch: The screen image is composed of a number of pixel elements. A team pixel is the smallest unit of display screen (the word comes from combination of picture and elements). Each pixel is composed of three phosphor dots, Red, Green and Blue.

Now the Dot- pitch is the distance between the phosphor dots that make up a single pixel. The Dot-pitch of color monitors for PCs ranges from about 0.15 mm to 0.30 mm.

Resolution: The maximum under of points that can be displayed without overlap on a monitor screen is referred to as the resolution. The resolution of a monitor indicates how densely the pixels (a single point in a graphic image) are packed. On color monitors, each pixel is actually composed of three dots – a Red, Green and Blue. Ideally the three points should converge at the same point. Resolution indicates the quality of monitor. Greater the number of pixels, better the resolution and sharper the image. Actually the resolution is determined by the video controller or video adapter card. IBM has provided following adapter cards:

Video Graphics Array (VGA) ------------------------640 × 480 pixels

Super Visual Graphics Array (SVGA) --------------800 × 600 pixels

----------------1024 × 768 pixels

Extended Graphics Array (XGA) -----------------------1152 × 864 pixels

iv. Refresh Rate: Refresh Rate is the number of times per second at which each pixel on a screen is refreshed. Display monitors must be redrawn many times per second. The refresh rate for a monitor is measured in hertz (HZ) or cycles/second. Generally monitors refresh rates are 60 HZ or 70 HZ. The faster the refresh rate the less the monitor flickers.

Classification of Monitors based on color

Monitors are classified into three categories according to its display color

Monochrome monitors

Gray-scale monitors

Color monitors

Monochrome: Monochrome monitors actually display two colors, one for the background and another for foreground. The color can be black (background) and white (foreground), black (background) and green (foreground).

Grey-scale: Gray-scale is a special type of monochrome monitor capable of displaying different shades of gray. Background color is usually white in such monitors.

Color: Color monitors can display from 1 to 16 million different colors. These monitors are sometimes called RGB (Red, Green, and Blue) monitors because three primary colors Red, Green and Blue are used to make other colors. An RGB monitor consists of cathode ray tube with three-electron guns-one each for Red, Green and Blue at one end and screen at the other end.

Color and gray-scale monitors are often classified by the number of they use to represent each pixel. For example, a 14-bit monitor represents each pixel with 24 bits. The more bits per pixel, the more colors and shades the monitor can display.

<u>Classification of Monitors based on Technology:</u>

Cathode Ray Tube (CRT)

Flat Panel Display

a. Cathode Ray Tube (CRT)

A traditional picture tube is like a big glass bottle. There are electron guns in the narrow end. They fire towards the large flat surface facing the user. The inside of the glass surface is coated with tiny phosphorus dots. There are arrange in groups of three- Red, Blue and Green (for color monitor) phosphorus dot. These dots light up or grow when hit by electrons from the electron gun. The more powerful beam is, the brighter they get. The electron beams are guided by electromagnets, where bend the beams, so they hit the exact desired phosphorus dot. Data imaged is product by moving the electron beam across the phosphor coated screen. Phosphor coating can be made to glow with different intensities by varying the strength of the electron beam, which finally forms character on screen. CRT monitor contains a shadow mask, which is fine mesh made of up metal., fitted to the shape and size of the screen. The holes in the shadow mask's mesh are used to align the electron beams, to insure that they strike precisely the correct phosphorus dot. Mostly, these holes are arranged in triangles.

CRT monitor are popular with desktop computers. These are available in monochrome and color monitors. The CRT has display screen of 25 lines of 80 characters each.

Types of CRT monitors:

There are two types of CRT monitors:

i. Composite CRT monitors

Composite CRT monitor is used only one electron gun to control the intensity of all these phosphorus dots in each pixel.

ii. **RGB CRT monitors**

RGB CRT monitor uses three individual guns, one for its dot, to control the intensity. Each of the sub dots are hit by its own electron gun that is why these monitors give sharper picture.

Advantages of CRT:

a) CRT monitors have wider viewing angle.

b) CRT monitors are cheaper and durable.

c) CRT monitors give sharp and crisp images.

d) CRT monitors resolution is adjustable.

Disadvantage of CRT:

a) As the phosphor dots start to fade after sometime they are hit by the electron gun, they need to be refreshed again.

b) The screen may flicker causing eyestrain.

c) They are bulky and heavy.

d) They consume high power.

b. Flat panel display

A CRT screen is reliable but it is bulky & consumes a lot of power so it is not used for portable computers. For small computers, flat panel display is used. The most common type of flat panel display are:

i. Liquid crystal display (LCD)

ii. Gas Plasma Display (GPD)

i. Liquid Crystal Display (LCD)

LCD produces images by aligning molecular crystals. When a voltage is applied, the crystals line up in a way that blocks light from passing through them and the absence of light is seen as characters on the screen. The LCD screen is flat, since it does not have picture tube. Instead the screen image is generated on a flat plastic disk. Thus LCD screens are much thinner, soft and don’t flicker as compared to CRT.

Advantages of LCD

LCD is light weight, flat, thin and require less space.

LCD does not require periodic refreshing.

LCD emits zero radiation thereby avoiding eye strain.

LCD consumes less power.

Disadvantages of LCD

LCD have smaller viewing angle so picture is best viewed when the person is in straight position from the centre of monitor.

The liquid crystals do not emit light so the images are less sharp.

There is a need of backlight setting to enhance sharpness of images.

Resolution is normally set.

ii. Gas Plasma Display (GPD)

One of the advancement in flat panel display is GPD. Gas Plasma display offers flicker free viewing and has higher contrast than LCDs. GPD contains ionized gas (Neon or Xenon) in between two glass plates. Among the two glass plates, one has serial of vertical wires or electrodes while other has series of horizontal wires. When the two placed together, the intersection of horizontal and vertical wire identifies a pixel. When current is applied through appropriate vertical and horizontal lines, the gas at the pixel emits light. Thus characters are formed by glowing combination of appropriate pixels.

CPU Architecture

Dataflow architecture topics

Static dataflow machines

Designs that use conventional memory addresses as data dependency tags are called static dataflow machines. These machines did not allow multiple instances of the same routines to be executed simultaneously because the simple tags could not differentiate between them.

Designs that use content-addressable memory (CAM) are called dynamic dataflow machines. They use tags in memory to facilitate parallelism.

Compiler

Normally, compilers analyze program source code for data dependencies between instructions in order to better organize the instruction sequences in the binary output files. The instructions are organized sequentially but the dependency information itself is not recorded in the binaries. Binaries compiled for a dataflow machine contain this dependency information.

A dataflow compiler records these dependencies by creating unique tags for each dependency instead of using variable names. By giving each dependency a unique tag, it allows the non-dependent code segments in the binary to be executed out of order and in parallel.

Programs

Programs are loaded into the CAM of a dynamic dataflow computer. When all of the tagged operands of an instruction become available (that is, output from previous instructions and/or user input), the instruction is marked as ready for execution by an execution unit.

This is known as activating or firing the instruction. Once an instruction is completed by an execution unit, its output data is stored (with its tag) in the CAM. Any instructions that are dependent upon this particular datum (identified by its tag value) are then marked as ready for execution. In this way, subsequent instructions are executed in proper order, avoiding race conditions. This order may differ from the sequential order envisioned by the human programmer, the programmed order.

Instructions

An instruction, along with its required data operands, is transmitted to an execution unit as a packet, also called an instruction token. Similarly, output data is transmitted back to the CAM as a data token. The packetization of instructions and results allows for parallel execution of ready instructions on a large scale.

Dataflow networks deliver the instruction tokens to the execution units and return the data tokens to the CAM. In contrast to the conventional von Neumann architecture, data tokens are not permanently stored in memory, rather they are transient messages that only exist when in transit to the instruction storage.

Classification of primary memory

In computing, memory refers to the physical devices used to store programs (sequences of instructions) or data (e.g. program state information) on a temporary or permanent basis for use in a computer or other digital electronic device. The term primary memory is used for the information in physical systems which function at high-speed (i.e. RAM), as a distinction from secondary memory, which are physical devices for program and data storage which are slow to access but offer higher memory capacity. Primary memory stored on secondary memory is called "virtual memory". An archaic synonym for memory is store.[1]

The term "memory", meaning primary memory is often associated with addressable semiconductor memory, i.e. integrated circuits consisting of silicon-based transistors, used for example as primary memory but also other purposes in computers and other digital electronic devices. There are two main types of semiconductor memory: volatile and non-volatile. Examples of non-volatile memory are flash memory (sometimes used as secondary, sometimes primary computer memory) and ROM/PROM/EPROM/EEPROM memory (used for firmware such as boot programs). Examples of volatile memory are primary memory (typically dynamic RAM, DRAM), and fast CPU cache memory (typically static RAM, SRAM, which is fast but energy-consuming and offer lower memory capacity per area unit than DRAM) .

Most semiconductor memory is organized into memory cells or bitable flip-flops, each storing one bit (0 or 1). Flash memory organization includes both one bit per memory cell and multiple bits per cell (called MLC, Multiple Level Cell). The memory cells are grouped into words of fixed word length, for example 1, 2, 4, 8, 16, 32, 64 or 128 bit. Each word can be accessed by a binary address of N bit, making it possible to store 2 raised by N words in the memory. This implies that processor registers normally are not considered as memory, since they only store one word and do not include an addressing mechanism.

The term storage is often used to describe secondary memory such as tape, magnetic disks and optical discs (CD-ROM and DVD-ROM).

Secondary memory

Secondary memory, also known as **auxiliary storage**, **secondary storage**, **secondary memory** or **external memory**, is used to store a large amount of data at lesser cost per byte than primary memory. They are two orders of magnitude less expensive than primary storage. Also Secondary storage does not lose the data when the device is powered down—it is non-volatile. Another difference from primary storage in that it is not directly accessible by the CPU, they are accessed via the input/output channels. The most common auxiliary memory devices currently used in computer systems are flash memories, magnetic disks and optical memories. Other components that were used previously are magnetic tapes, magnetic drums, magnetic bubble memory etc. Flash memories are the latest addition to the family; they are much faster compared to its predecessors as they don't have any moving parts. In few of the laptops (Apple’s MAC Book Air) Solid State Drives (SSDs) made from flash memory have started replacing the magnetic disc based Hard Disk Drives (HDDs).

Auxiliary (Secondary) Memories

Flash memory: An electronic non-volatile computer storage device that can be electrically erased and reprogrammed, and works without any moving parts.

Optical disc: It’s a storage medium from which data is read and to which it is written by lasers. Optical disks can store much more data—up to 6 gigabytes (6 billion bytes)—than most portable magnetic media, such as floppies. There are three basic types of optical disks: CD-ROM (Read only), WORM (Write-Once Read-Many) & EO (Erasable Optical disks).

Magnetic Disk: A magnetic disk is a circular plate constructed of metal or plastic coated with magnetized material. Both sides of the disk are used and several disks may be stacked on one spindle with read/write heads available on each surface. Bits are stored in magnetized surface in spots along concentric circles called tracks. Tracks are commonly divided into sections called sectors. Disk that are permanently attached and cannot removed by occasional user are called Hard disk. A disk drive with removable disks is called floppy disks.

Magnetic tapes: A magnetic tape transport consists of electric, mechanical and electronic components to provide the parts and control mechanism for a magnetic tape unit. The tape itself is a strip of plastic coated with a magnetic recording medium. Bits are recorded as magnetic spots on tape along several tracks. Seven or Nine bits are recorded to form a character together with a parity bit R/W heads are mounted in each track so that data can be recorded and read as a sequence of characters.

USB flash drive

A **USB flash drive** is a data storage device that includes flash memory with an integrated Universal Serial Bus (USB) interface. USB flash drives are typically removable and rewritable, and physically much smaller than an optical disc. Most weigh less than 30 grams (1.1 oz).As of September 2011, drives of up to 256 gigabytes (GB) are available. A one-terabyte (TB) drive was unveiled at the 2013 Consumer Electronics Show and will be available during 2013. Storage capacities as large as 2 TB are planned, with steady improvements in size and price per capacity expected.[4] Some allow up to 100,000 write/erase cycles, depending on the exact type of memory chip used, and a 10-year shelf storage time.

USB flash drives are often used for the same purposes for which floppy disks or CD-ROMs were used, i.e., for storage, back-up and transfer of computer files. They are smaller, faster, have thousands of times more capacity, and are more durable and reliable because they have no moving parts. Until about 2005, most desktop and laptop computers were supplied with floppy disk drives in addition to USB ports, but floppy disk drives have been abandoned due to their lower capacity compared to USB flash drives.

USB flash drives use the USB mass storage standard, supported natively by modern operating systems such as Linux, Mac OS X, Windows, and other Unix-like systems, as well as many BIOS boot ROMs. USB drives with USB 2.0 support can store more data and transfer faster than much larger optical disc drives like CD-RW or DVD-RW drives and can be read by many other systems such as the Xbox 360, PlayStation 3, DVD players and in a number of handheld devices such as smart phones and tablet computers.

Serial Port

In computing, a **serial port** is a serial communication physical interface through which information transfers in or out one bit at a time (in contrast to a parallel).[1] Throughout most of the history of personal computers, data was transferred through serial ports to devices such as modems, terminals and various peripherals.

While such interfaces as Ethernet, FireWire, and USB all send data as a serial stream, the term "serial port" usually identifies hardware more or less compliant to the RS-232 standard, intended to interface with a modem or with a similar communication device.

Parallel port

A **parallel port** is a type of interface found on computers (personal and otherwise) for connecting peripherals. In computing, a parallel port is a parallel communication physical interface. It is also known as a **printer port** or Getronics port. It was a de facto industry standard for many years, and was finally standardized as IEEE 1284 in the late 1990s, which defined a bi-directional version of the port. Today, the parallel port interface is seeing decreasing use because of the rise of Universal Serial Bus (USB) and FireWire (IEEE 1394) devices, along with network printing using Ethernet.

The parallel port interface was originally known as the **Parallel Printer Adapter** on IBM PC-compatible computers. It was primarily designed to operate a line printer that used IBM's 8-bit extended ASCII character set to print text, but could also be used to adapt other peripherals. Graphical printers, along with a host of other devices, have been designed to communicate with the system.

Universal Serial Bus (**USB**)

Universal Serial Bus (**USB**) is an industry standard developed in the mid-1990s that defines the cables, connectors and communications protocols used in a bus for connection, communication and power supply between computers and electronic devices.

USB was designed to standardize the connection of computer peripherals (including keyboards, pointing devices, digital cameras, printers, portable media players, disk drives and network adapters) to personal computers, both to communicate and to supply electric power. It has become commonplace on other devices, such as smart phones, PDAs and video game consoles. USB has effectively replaced a variety of earlier interfaces, such as serial and parallel ports, as well as separate power chargers for portable devices.

As of 2008, approximately six billion USB ports and interfaces were in the global marketplace, and about 2 billion were being sold each year.

Protocol   Week No: 08/09

The **Internet protocol suite** is the networking model and a set of communications used for the Internet and similar networks. It is commonly known as **TCP/IP**, because its most important protocols, the Transmission Control Protocol (TCP) and the Internet Protocol (IP) were the first networking protocols defined in this standard. It is occasionally known as the **DOD model** due to the foundational influence of the ARPANET in the 1970s (operated by DARPA, an agency of the United States Department of Defense).

TCP/IP provides end-to-end connectivity specifying how data should be formatted, addressed, transmitted, routed and received at the destination. It has four abstraction layers which are used to sort all related protocols according to the scope of networking involved. From lowest to highest, the layers are:

- The link layer contains communication technologies for a single network segment (link) of a local area network.
- The internet layer (IP) connects independent networks, thus establishing internetworking.
- The transport layer handles host-to-host communication.

- The application layer contains all protocols for specific data communications services on a process-to-process level. For example, the Hypertext Transfer Protocol (HTTP) specifies the web browser communication with a web server.

The TCP/IP model and related protocols are maintained by the Internet Engineering Task Force (IETF).

The **Transmission Control Protocol** (**TCP**) is one of the core protocols of the Internet protocol suite (IP), and is so common that the entire suite is often called TCP/IP. TCP provides reliable, ordered, error-checked delivery of a stream of octets between programs running on computers connected to a local area network, intranet or the public Internet. It resides at the transport layer.

Web browsers use TCP when they connect to servers on the World Wide Web, and it is used to deliver email and transfer files from one location to another.

Applications that do not require the reliability of a TCP connection may instead use the connectionless User Datagram Protocol (UDP), which emphasizes low-overhead operation and reduced latency rather than error checking and delivery validation.

Internet service provider

An **Internet service provider** (**ISP**, also called **Internet access provider**) is a business or organization that offers user's access to the Internet and related services. Many but not all ISPs are telephone companies or other telecommunication providers. They provide services such as Internet, Internet transit, domain name registration and hosting, dial-up access, leased line access and collocation. Internet service providers may be organized in various forms, such as commercial, community, non-profit, or otherwise privately.

Network service provider (NSP)

A **network service provider** (**NSP**) is a business or organization that sells bandwidth or network access by providing direct Internet access to the Internet and usually access to its network access points (NAPs) For such a reason, network service providers are sometimes referred to as backbone providers or internet providers.

Network service providers may consist of telecommunications companies, data carriers, wireless communications providers, Internet, and cable television operators offering high-speed Internet access.

A few information technology companies are also emerging the NSP market, notably IBM, EDS, CSC, Vanco and Atos Origin. This is due to the technological convergence of information and communications technology.

Concept Of site & Pages   week No: 09

A **website**, also written as **Web site**, **web site**, or simply **site** is a set of related web pages served from a single web domain. A website is hosted on at least one web server, accessible via a network such as the Internet or a private local through an Internet address known as a Uniform Resource Locator. All publicly accessible websites collectively constitute the World Wide Web.

A webpage is a document, typically written in plain text interspersed with formatting instructions of Hypertext Markup Language (HTML, XHTML). A webpage may incorporate elements from other websites with suitable markup anchors.

WebPages are accessed and transported with the Hypertext Transfer Protocol(HTTP), which may optionally employ encryption (HTTP Secure, HTTPS) to provide security and privacy for the user of the webpage content. The user's application, often a web browser, renders the page content according to its HTML markup instructions onto a display terminal.

The pages of a website can usually be accessed from a simple Uniform Resource Locator (URL) called the web address. The URLs of the pages organize them into a hierarchy, although hyper linking between them conveys the reader's perceived site structure and guides the reader's navigation of the site which generally includes a home page with most of the links to the site's web, and a supplementary about, contact and link page.

Some websites require a subscription to access some or all of their content. Examples of subscription websites include many business sites, parts of news websites, academic journal websites, gaming websites, file-sharing websites, message boards, web-based email, social networking websites, websites providing real-time stock data, and websites providing various other services (e.g., websites offering storing and/or sharing of images, files and so forth).

Hypertext Markup Language (**HTML**)

Hypertext Markup Language (**HTML**) is the main markup language for creating web pages and other information that can be displayed in a web browser.

HTML is written in the form of HTML elements consisting of tags enclosed in angle (like <html>), within the web page content. HTML tags most commonly come in pairs like <h1> and </h1>, although some tags represent empty elements and so are unpaired, for example <img>. The first tag in a pair is the start tag, and the second tag is the end tag (they are also called opening tags and closing tags). In between these tags web designers can add text, further tags, comments and other types of text-based content.

The purpose of a web browser is to read HTML documents and compose them into visible or audible web pages. The browser does not display the HTML tags, but uses the tags to interpret the content of the page.

HTML elements form the building blocks of all websites. HTML allows images and objects to be embedded and can be used to create interactive forms. It provides a means to create structured documents by denoting structural semantics for text such as headings, paragraphs, lists, links, quotes and other items. It can embed scripts written in languages such as JavaScript which affect the behavior of HTML web pages.

Web browsers can also refer to Cascading Style Sheets (CSS) to define the appearance and layout of text and other material. The W3C, maintainer of both the HTML and the CSS standards, encourages the use of CSS over explicit presentational HTML markup.

DHTML

Dynamic HTML, or **DHTML**, is an umbrella term for a collection of technologies used together to create interactive and animated web sites by using a combination of a static markup language (such as HTML), a scripting language (such as JavaScript), a presentation definition language (such as CSS), and the Document Object Model.

DHTML allows scripting languages to change variables in a web page's definition language, which in turn affects the look and function of otherwise "static" HTML page content, after the page has been fully loaded and during the viewing process. Thus the dynamic characteristic of DHTML is the way it functions while a page is viewed, not in its ability to generate a unique page with each page load.

By contrast, a dynamic web page is a broader concept, covering any web page generated differently for each user, load occurrence, or specific variable values. This includes pages created by client-side scripting, and ones created by server-side scripting (such as PHP, Perl, JSP or ASP.NET) where the web server generates content before sending it to the client.

DHTML is differentiated from AJAX by the fact that a DHTML page is still request/reload-based. With DHTML, there may not be any interaction between the client and server after the page is loaded; all processing happens in JavaScript on the client side. By contrast, an AJAX page uses features of DHTML to initiate a request (or 'sub request') to the server to perform actions such as loading more content.

XML

Extensible Markup Language (**XML**) is a markup language that defines a set of rules for encoding documents in a format that is both human-readable and machine. It is defined in the XML 1.0 Specification produced by theW3C, and several other related specifications, all gratis open standards.

The design goals of XML emphasize simplicity, generality, and usability over the Internet. It is a textual data format with strong support via Unicode for the languages of the world. Although the design of XML focuses on documents, it is widely used for the representation of arbitrary data structures, for example in web.

Many application programming interfaces (APIs) have been developed to aid software developers with processing XML data, and several systems exist to aid in the definition of XML-based languages.

As of 2009, hundreds of document formats using XML syntax have been developed, including RSS, Atom, SOAP, and XHTML. XML-based formats have become the default for many office-productivity tools, including Microsoft Office(Office Open XML), OpenOffice.org and Libra Office (Open Document), and Apple's work. XML has also been employed as the base language for communication, such as XMPP.

INFORMATION Week No: 10/11

Information, in its most restricted technical sense, is a sequence of symbols that can be interpreted as a message. Information can be recorded as signs, or transmitted as signals. Information is any kind of event that affects the state of a dynamic system that can interpret the information.

Conceptually, information is the message (utterance or expression) being conveyed. Therefore, in a general sense, information is "Knowledge communicated or received concerning a particular fact or circumstance "Information cannot be predicted and resolves uncertainty. The uncertainty of an event is measured by its probability of occurrence and is inversely proportional to that. The more uncertain an event is more information is required to resolve uncertainty of that event. The amount of information is measured in bits.
Example: information in one "fair" coin flip: $\log_2 (2/1) = 1$ bit, and in two fair coin flips is $\log_2(4/1) = 2$ bits.
The concept that information is the message has different meanings in different

contexts. Thus concept of information becomes closely related to notions of constraint, communication, control, data, form, instruction, knowledge, meaning, understanding, mental stimuli, pattern, perception, representation, and entropy.

Difference between data information

	Data	**Information**
Meaning:	Data is raw, unorganized facts that need to be processed. Data can be something simple and seemingly random and useless until it is organized.	When data is processed, organized, structured or presented in a given context so as to make it useful, it is called Information.
Example:	Each student's test score is one piece of data	The class' average score or the school's average score is the information that can be concluded from the given data.
Definition:	Latin 'datum' meaning "that which is given". Data was the plural form of datum singular (M150 adopts the general use of data as singular. Not everyone agrees.)	Information is interpreted data.

DATA Concept:

Logical Data Independence

What do you mean by logical data independence? The ability to change the logical schema without changing the external schema or application programs is called as Logical Data Independence. OR The ability to change the logical schema without having to change the external schema.

Examples: The addition or removal of new entities, attributes, or relationships to the conceptual schema should be possible without having to change existing external schemas or having to rewrite existing application programs.

Physical Data Independence

Modifying indexes. Using different file organizations or storage structures. Switching from one access method to another. Using different data structures. Using new storage devices. The ability to change the physical schema without changing the logical schema is called as Physical Data Independence. Changes in the physical schema may include.What do you mean by Physical Data Independence?

Examples: A change to the internal schema, such as using different file organization or storage structures, storage devices, or indexing strategy, should be possible without having to change the conceptual or external schemas.

Data Processing

Data processing is "the collection and manipulation of items of data to produce meaningful information."

Data processing is distinct from word processing, which manipulates text rather than data. It is a subset of information processing, "the change (processing) of information in any manner detectable by an observer."

DATA Security

Data Security means protecting a database from destructive forces and the unwanted actions of unauthorized users.

Disk encryption refers to encryption technology that encrypts data on a hard disk drive. Disk encryption typically takes form in either software (see disk encryption software] or hardware (see disk encryption hardware). Disk encryption is often referred to as on-the-fly encryption ("OTFE") or transparent encryption.

In information technology, a **backup**, or the process of **backing up**, refers to the copying and archiving of computer data so it may be used to restore the original after a data loss event. The verb form is to **back up** in two words, whereas the noun is backup.

Backups have two distinct purposes. The primary purpose is to recover data after its loss, be it by data deletion or corruption. Data loss can be a common experience of computer users. A 2008 survey found that 66% of respondents had lost files on their home PC. The secondary purpose of backups is to recover data from an earlier time, according to a user-defined data retention policy, typically configured within a backup application for how long copies of data are required. Though backups popularly represent a simple form of disaster recovery, and should be part of a disaster recovery plan, by themselves, backups should not alone be considered disaster recovery. One reason for this is that not all backup systems or backup applications are able to reconstitute a computer system or other complex configurations such as a computer cluster, active directory servers, or a database server, by restoring only data from a backup.

Since a backup system contains at least one copy of all data worth saving, the data storage requirements can be significant. Organizing this storage space and managing the backup process can be complicated undertaking. A data repository model can be used to provide structure to the storage. Nowadays, there are many different types of data storage devices that are useful for making backups. There are also many different ways in which these devices can be arranged to provide geographic redundancy, data security, and portability.

Before data is sent to its storage location, it is selected, extracted, and manipulated. Many different techniques have been developed to optimize the backup procedure. These include optimizations for dealing with open files and live data sources as well as compression, encryption, and de-duplication, among others. Every backup scheme should include dry runs that validate the reliability of the data being backed up. It is important to recognize the limitations and human factors involved in any backup scheme.

Data Masking

Data masking of structured data is the process of obscuring (masking) specific data within a database table or cell to ensure that data security is maintained and sensitive information is not exposed to unauthorized personnel. This may include masking the data from users (for example so banking customer representatives can only see the last 4 digits of a customer's national identity number), developers (who need real production data to test new software releases but should not be able to see sensitive financial data), outsourcing vendors, etc.

RESTORETION

In data management, restore is a process that involves copying backup files from secondary storage (tape, zip disk or other backup media) to hard disk. A restore is performed in order to return data to its original condition if files have become damaged, or to copy or move data to a new location.

Flowchart

Week No:12/14

A **flowchart** is a type of diagram that represents an algorithm or process, showing the steps as boxes of various kinds, and their order by connecting them with arrows. This diagrammatic representation illustrates a solution to a given problem. Process operations are represented in these boxes, and arrows; rather, they are implied by the sequencing of operations. Flowcharts are used in analyzing, designing, documenting or managing a process or program in various fields.

Flowcharts are used in designing and documenting complex processes or programs. Like other types of diagrams, they help visualize what is going on and thereby help the viewer to understand a process, and perhaps also find flaws, bottlenecks, and other less-obvious features within it. There are many different types of flowcharts, and each type has its own repertoire of boxes and notational conventions. The two most common types of boxes in a flowchart are:

- a processing step, usually called activity, and denoted as a rectangular box
- a decision, usually denoted as a diamond.

A flowchart is described as "cross-functional" when the page is divided into different swim lanes describing the control of different organizational units. A symbol appearing in a particular "lane" is within the control of that organizational unit. This technique allows the author to locate the responsibility for performing an action or making a decision correctly, showing the responsibility of each organizational unit for different parts of a single process.

Flowcharts depict certain aspects of processes and they are usually complemented by other types of diagram. For instance, Kaoru Ishikawa defined the flowchart as one of the seven basic tools of quality control, next to the histogram, Pareto chart, check, control chart, cause-and-effect diagram, and the scatter diagram. Similarly, in UML, a standard concept-modeling notation used in software development, the activity diagram, which is a type of flowchart, is just one of many different diagram types.

Nassi-Shneiderman diagrams are an alternative notation for process flow.

Common alternate names include: flowchart, process flowchart, functional flowchart, process map, process chart, functional process chart, business process model, process model, process flow diagram, work flow diagram, business flow diagram. The terms "flowchart" and "flow chart" are used interchangeably.

The first structured method for documenting process flow, the "flow process chart", was introduced by Frank Gilbert to members of the American Society of Mechanical Engineers (ASME) in 1921 in the presentation “Process Charts—First Steps in Finding the One Best Way”. Gilbreth's tools quickly found their way into industrial curricula. In the early 1930s, an industrial engineer, Allan H. Mogensen began training business people in the use of some of the tools of industrial engineering at his Work Simplification Conferences in Lake Placid, New York.

Types of Flow chart

Sterneckert (2003) suggested that flowcharts can be modeled from the perspective of different user groups (such as managers, system analysts and clerks) and that there are four general types:

- Document flowcharts, showing controls over a document-flow through a system
- Data flowcharts, showing controls over a data-flow in a system
- System flowcharts showing controls at a physical or resource level
- Program flowchart, showing the controls in a program within a system

Notice that every type of flowchart focuses on some kind of control, rather than on the particular flow itself.

However there are several of these classifications. For example Andrew Veronis (1978) named three basic types of flowcharts: the system flowchart, the general flowchart, and the detailed flowchart. That same year Marilyn Bohl (1978) stated "in practice, two kinds of flowcharts are used in solution planning: system flowcharts and program flowcharts...". More recently Mark A. Fryman (2001) stated that there are more differences: "Decision flowcharts, logic flowcharts, systems flowcharts, product flowcharts, and process flowcharts are just a few of the different types of flowcharts that are used in business and government"

In addition, many diagram techniques exist that are similar to flowcharts but carry a different name, such as UML activity diagrams.

Overview of C Language

Why use C?

C has been used successfully for every type of programming problem imaginable from operating systems to spreadsheets to expert systems - and efficient **compilers** are available for machines ranging in power from the **Apple** Macintosh to the **Cray** supercomputers. The largest measure of C's success seems to be based on purely practical considerations:

1. the portability of the compiler;
2. the standard library concept;
3. a powerful and varied repertoire of operators;
4. an elegant syntax;
5. ready access to the hardware when needed;
6. and the ease with which applications can be optimized by hand-coding isolated procedures

C is often called a "Middle Level" programming language. This is not a reflection on its lack of programming power but more a reflection on its capability to access the system's low level functions. Most high-level languages (e.g. Fortran) provides everything the programmer might want to do already built into the language. A low level language (e.g. **assembler**) provides nothing other than access to the machines basic instruction set. A middle level language, such as C, probably doesn't supply all the constructs found in high-languages - but it provides you with all the building blocks that you will need to produce the results you want!

Uses of C

C was initially used for system development work, in particular the programs that make-up the operating system. Why use C? Mainly because it produces code that runs nearly as fast as code written in assembly language. Some examples of the use of C might be:

1. Operating Systems
2. Language Compilers
3. Assemblers

4. Text Editors
5. Print Spoolers
6. Network Drivers
7. Modern Programs
8. Data Bases
9. Language Interpreters
10. Utilities

In recent years C has been used as a general-purpose language because of its popularity with programmers. It is not the world's easiest language to learn and you will certainly benefit if you are not learning C as your first programming language! C is trendy (I nearly said sexy) - many well established programmers are switching to C for all sorts of reasons, but mainly because of the portability that writing standard C programs can offer.

A Brief History of C

C is a general-purpose language which has been closely associated with the **UNIX** operating system for which it was developed - since the system and most of the programs that run it are written in C.

Many of the important ideas of C stem from the language **BCPL**, developed by **Martin Richards**. The influence of BCPL on C proceeded indirectly through the language **B**, which was written by **Ken Thompson** in 1970 at Bell Labs, for the first UNIX system on a **DEC** PDP-7. **BCPL** and **B** are "type less" languages whereas C provides a variety of data types.

In 1972 **Dennis Ritchie** at Bell Labs writes C and in 1978 the publication of The C Programming Language by **Kernighan & Ritchie** caused a revolution in the computing world.

In 1983, the American National Standards Institute (ANSI) established a committee to provide a modern, comprehensive definition of C. The resulting definition, the ANSI standard, or "ANSI C", was completed late 1988.

C for Personal Computers

With regards to personal computers Microsoft C for IBM (or clones) PC's. and Borland's C are seen to be the two most commonly used systems. However, the latest version of Microsoft C is now considered to be the most powerful and efficient C compiler for personal computers.

C - Using Constants

A C constant is usually just the written version of a number. For example 1, 0, 5.73, 12.5e9. We can specify our constants in octal or hexadecimal, or force them to be treated as long integers.

- Octal constants are written with a leading zero - 015.
- Hexadecimal constants are written with a leading 0x - 0x1ae.
- Long constants are written with a trailing L - 890L.

Character constants are usually just the character enclosed in single quotes; 'a', 'b', 'c'. Some characters can't be represented in this way, so we use a 2 character sequence as follows.

```
'\n'    newline
'\t'    tab
'\\'    backslash
'\''    single quote
'\0'    null ( Used automatically to terminate character string )
```

In addition, a required bit pattern can be specified using its octal equivalent.

'\044' produces bit pattern 00100100.

Character constants are rarely used, since string constants are more convenient. A string constant is surrounded by double quotes eg "Brian and Dennis". The string is actually stored as an array of characters. The null character '\0' is automatically placed at the end of such a string to act as a string terminator.

A character is a different type to a single character string. This is important poing to note.

Defining Constants

ANSI C allows you to declare *constants*. When you declare a constant it is a bit like a variable declaration except the value cannot be changed.

The *const* keyword is to declare a constant, as shown below:

```
int const a = 1;
const int a =2;
```

Note:

- You can declare the *const* before or after the type. Choose one an stick to it.
- It is usual to initialise a *const* with a value as it cannot get a value *any other way*.

The preprocessor *#define* is another more flexible (see Preprocessor Chapters) method to define*constants* in a program.

```
#define TRUE          1
#define FALSE         0
#define NAME_SIZE     20
```

Here TRUE, FALSE and NAME_SIZE are constant

You frequently see const declaration in function parameters. This says simply that the function is**not** going to change the value of the parameter.

The following function definition used concepts we have not met (see chapters on functions, strings, pointers, and standard libraries) but for completenes of this section it is is included here:

```
void strcpy(char *buffer, char const *string)
```

The enum Data type

enum is the abbreviation for ENUMERATE, and we can use this keyword to declare and initialize a sequence of integer constants. Here's an example:

```
enum colors {RED, YELLOW, GREEN, BLUE};
```

I've made the constant names uppercase, but you can name them which ever way you want.

Here, *colors* is the name given to the set of constants - the name is optional. Now, if you don't assign a value to a constant, the default value for the first one in the list - *RED* in our case, has the value of *0*. The rest of the undefined constants have a value 1 more than the one before, so in our case, *YELLOW* is *1*, *GREEN* is *2* and *BLUE* is *3*.

But you can assign values if you wanted to:

```
enum colors {RED=1, YELLOW, GREEN=6, BLUE };
```

Now RED=1, YELLOW=2, GREEN=6 and BLUE=7.

The main advantage of *enum* is that if you don't initialize your constants, each one would have a unique value. The first would be zero and the rest would then count upwards.

You can name your constants in a weird order if you really wanted...

```
#include <stdio.h>

int main() {
enum {RED=5, YELLOW, GREEN=4, BLUE};

printf("RED = %d\n", RED);
printf("YELLOW = %d\n", YELLOW);
printf("GREEN = %d\n", GREEN);
printf("BLUE = %d\n", BLUE);
return 0;
}
```

```
This will produce following results

RED = 5
YELLOW = 6
GREEN = 4
BLUE = 5
```

C - Variable Types

A variable is just a named area of storage that can hold a single value (numeric or character). The C language demands that you declare the name of each variable that you are going to use and its type, or class, before you actually try to do anything with it.

The Programming language C has two main variable types

- Local Variables
- Global Variables

Local Variables

- Local variables scope is confined within the block or function where it is defined. Local variables must always be defined at the top of a block.
- When a local variable is defined - it is not initalised by the system, you must initalise it yourself.
- When execution of the block starts the variable is available, and when the block ends the variable 'dies'.

Check following example's output

```
main()
{
int i=4;
int j=10;

i++;

if (j > 0)
{
/* i defined in 'main' can be seen */
printf("i is %d\n",i);
}

if (j > 0)
{
/* 'i' is defined and so local to this block */
int i=100;
printf("i is %d\n",i);
}/* 'i' (value 100) dies here */

printf("i is %d\n",i); /* 'i' (value 5) is now visable.*/
}
```

```
This will generate following output
i is 5
i is 100
i is 5
```

Here **++** is called incremental operator and it increase the value of any integer variable by 1. Thus **i++** is equivalent to *i = i + 1;*

You will see **--** operator also which is called decremental operator and it idecrease the value of any integer variable by 1. Thus **i--** is equivalent to *i = i - 1;*

Global Variables

Global variable is defined at the top of the program file and it can be visible and modified by any function that may reference it.

Global variables are initalised automatically by the system when you define them!

Data Type	**Initialser**
int	0
char	'\0'
float	0
pointer	NULL

If same variable name is being used for global and local variable then local variable takes preference in its scope. But it is not a good practice to use global variables and local variables with the same name.

```
int i=4;          /* Global definition   */

main()
{
i++;          /* Global variable     */
func();
printf( "Value of i = %d -- main function\n", i );
}

func()
{
int i=10;     /* Local definition */
i++;          /* Local variable    */
printf( "Value of i = %d -- func() function\n", i );
}

This will produce following result
Value of i = 11 -- func() function
Value of i = 5 -- main function
```

i in **main** function is global and will be incremented to 5. **i** in **func** is internal and will be incremented to 11. When control returns to **main** the internal variable will die and and any reference to **i** will be to the global.

DATA Types

C has a concept of 'data types' which are used to define a variable before its use. The definition of a variable will assign storage for the variable and define the type of data that will be held in the location.

The value of a variable can be changed any time.

C has the following basic built-in datatypes.

- int
- float
- double
- char

Please note that there is not a boolean data type. C does not have the traditional view about logical comparison, but thats another story.

int - data type

int is used to define integer numbers.

```
{
int Count;
Count = 5;
}
```

float - data type

float is used to define floating point numbers.

```

{
float Miles;
Miles = 5.6;
}
```

double - data type

double is used to define BIG floating point numbers. It reserves twice the storage for the number. On PCs this is likely to be 8 bytes.

```
{
double Atoms;
Atoms = 2500000;
}
```

char - data type

char defines characters.

```
{
char Letter;
Letter = 'x';
}
```

Modifiers

The data types explained above have the following modifiers.

- short
- long
- signed
- unsigned

The modifiers define the amount of storage allocated to the variable. The amount of storage allocated is not cast in stone. ANSI has the following rules:

```

short int <=    int <= long int
float <= double <= long double

```

What this means is that a 'short int' should assign less than or the same amount of storage as an 'int' and the 'int' should be less or the same bytes than a 'long int'. What this means in the real world is:

```
Type Bytes             Range
---------------------------------------------------------------------
short int  2          -32,768 -> +32,767          (32kb)
unsigned short int  2               0 -> +65,535          (64Kb)
unsigned int  4               0 -> +4,294,967,295   ( 4Gb)
int  4   -2,147,483,648 -> +2,147,483,647   ( 2Gb)
long int  4   -2,147,483,648 -> +2,147,483,647   ( 2Gb)
signed char  1             -128 -> +127
unsigned char  1                0 -> +255
float  4
double  8
long double 12
```

These figures only apply to todays generation of PCs. Mainframes and midrange machines could use different figures, but would still comply with the rule above.

You can find out how much storage is allocated to a data type by using the **sizeof** operator discussed in Operator Types Session.

Here is an example to check size of memory taken by various datatypes.

```
int
main()
{
printf("sizeof(char) == %d\n", sizeof(char));
printf("sizeof(short) == %d\n", sizeof(short));
printf("sizeof(int) == %d\n", sizeof(int));
printf("sizeof(long) == %d\n", sizeof(long));
printf("sizeof(float) == %d\n", sizeof(float));
printf("sizeof(double) == %d\n", sizeof(double));
printf("sizeof(long double) == %d\n", sizeof(long double));
printf("sizeof(long long) == %d\n", sizeof(long long));

return 0;
}
```

Qualifiers

A type qualifier is used to refine the declaration of a variable, a function, and parameters, by specifying whether:

- The value of a variable can be changed.
- The value of a variable must always be read from memory rather than from a register

Standard C language recognizes the following two qualifiers:

- const
- volatile

The *const* qualifier is used to tell C that the variable value can not change after initialisation.

const float pi=3.14159;

Now *pi* cannot be changed at a later time within the program.

Another way to define constants is with the *#define* preprocessor which has the advantage that it does not use any storage

The volatile qualifier declares a data type that can have its value changed in ways outside the control or detection of the compiler (such as a variable updated by the system clock or by another program). This prevents the compiler from optimizing code referring to the object by storing the object's value in a register and re-reading it from there, rather than from memory, where it may have changed. You will use this qualifier once you will become expert in "C". So for now just proceed.

What are Arrays:

We have seen all baisc data types. In C language it is possible to make arrays whose elements are basic types. Thus we can make an array of 10 integers with the declaration.

```
int x[10];
```

The square brackets mean subscripting; parentheses are used only for function references. Array indexes begin at zero, so the elements of x are:

Thus Array are special type of variables which can be used to store multiple values of same data type. Those values are stored and accessed using subscript or index.

Arrays occupy consecutive memory slots in the computer's memory.

```
x[0], x[1], x[2], ..., x[9]
```

If an array has n elements, the largest subscript is n-1.

Multiple-dimension arrays are provided. The declaration and use look like:

```
int name[10] [20];
n = name[i+j] [1] + name[k] [2];
```

Subscripts can be arbitrary integer expressions. Multi-dimension arrays are stored by row so the rightmost subscript varies fastest. In above example **name** has 10 rows and 20 columns.

Same way, arrays can be defined for any data type. Text is usually kept as an array of characters. By convention in C, the last character in a character array should be a `\0' because most programs that manipulate character arrays expect it. For example, printf uses the `\0' to detect the end of a character array when printing it out with a `%s'.

Here is a program which reads a line, stores it in a buffer, and prints its length (excluding the newline at the end).

```
main( ) {
int n, c;
char line[100];
n = 0;
while( (c=getchar( )) != '\n' ) {
if( n < 100 )
line[n] = c;
n++;
}
printf("length = %d\n", n);
}
```

Array Initialization

- As with other declarations, array declarations can include an optional initialization

- Scalar variables are initialized with a single value
- Arrays are initialized with a list of values
- The list is enclosed in curly braces

```
int array [8] = {2, 4, 6, 8, 10, 12, 14, 16};
```

The number of initializers cannot be more than the number of elements in the array but it can be less in which case, the remaining elements are initialized to 0.if you like, the array size can be inferred from the number of initializers by leaving the square brackets empty so these are identical declarations:

```
int array1 [8] = {2, 4, 6, 8, 10, 12, 14, 16};
int array2 [] = {2, 4, 6, 8, 10, 12, 14, 16};
```

An array of characters ie string can be initialized as follows:

```
char string[10] = "Hello";
```

Operators

What is Operator? Simple answer can be given using expression *4 + 5 is equal to 9*. Here 4 and 5 are called operands and + is called operator. C language supports following type of operators.

- Arithmetic Operators
- Logical (or Relational) Operators
- Bitwise Operators
- Assignment Operators
- Misc Operators

Let’s have a look on all operators one by one.

Arithmetic Operators:

There are following arithmetic operators supported by C language:

Assume variable A holds 10 and variable B holds 20 then:

Show Examples

Operator	Description	Example
+	Adds two operands	A + B will give 30
-	Subtracts second operand from the first	A - B will give -10
*	Multiply both operands	A * B will give 200

/	Divide numerator by denumerator	B / A will give 2
%	Modulus Operator and remainder of after an integer division	B % A will give 0
++	Increment operator, increases integer value by one	A++ will give 11
--	Decrement operator, decreases integer value by one	A-- will give 9

Logical (or Relational) Operators:

There are following logical operators supported by C language

Assume variable A holds 10 and variable B holds 20 then:

Operator	Description	Example
==	Checks if the value of two operands is equal or not, if yes then condition becomes true.	(A == B) is not true.
!=	Checks if the value of two operands is equal or not, if values are not equal then condition becomes true.	(A != B) is true.
>	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.	(A > B) is not true.
<	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.	(A < B) is true.
>=	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.	(A >= B) is not true.
<=	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.	(A <= B) is true.

&&	Called Logical AND operator. If both the operands are non zero then then condition becomes true.	(A && B) is true.
||	Called Logical OR Operator. If any of the two operands is non zero then then condition becomes true.	(A || B) is true.
!	Called Logical NOT Operator. Use to reverses the logical state of its operand. If a condition is true then Logical NOT operator will make false.	!(A && B) is false.

Bitwise Operators:

A bitwise operator works on bits and performs bit by bit operation.

Assume if A = 60; and B = 13; Now in binary format they will be as follows:

A = 0011 1100

B = 0000 1101

A&B = 0000 1100

A|B = 0011 1101

A^B = 0011 0001

~A = 1100 0011

There are following Bitwise operators supported by C language

Operator	Description	Example
&	Binary AND Operator copies a bit to the result if it exists in both operands.	(A & B) will give 12 which is 0000 1100

|	Binary OR Operator copies a bit if it exists in either operand.	(A | B) will give 61 which is 0011 1101
^	Binary XOR Operator copies the bit if it is set in one operand but not both.	(A ^ B) will give 49 which is 0011 0001
~	Binary Ones Complement Operator is unary and has the efect of 'flipping' bits.	(~A) will give -60 which is 1100 0011
<<	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.	A << 2 will give 240 which is 1111 0000
>>	Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.	A >> 2 will give 15 which is 0000 1111

Assignment Operators:

There are following assignment operators supported by C language:

Operator	**Description**	**Example**
=	Simple assignment operator, Assigns values from right side operands to left side operand	C = A + B will assigne value of A + B into C
+=	Add AND assignment operator, It adds right operand to the left operand and assign the result to left operand	C += A is equivalent to C = C + A
-=	Subtract AND assignment operator, It subtracts right operand from the left operand and assign the result to left operand	C -= A is equivalent to C = C - A
*=	Multiply AND assignment operator, It multiplies right operand with the left operand and assign the result to left operand	C *= A is equivalent to C = C * A

/=	Divide AND assignment operator, It divides left operand with the right operand and assign the result to left operand	C /= A is equivalent to C = C / A
%=	Modulus AND assignment operator, It takes modulus using two operands and assign the result to left operand	C %= A is equivalent to C = C % A
<<=	Left shift AND assignment operator	C <<= 2 is same as C = C << 2
>>=	Right shift AND assignment operator	C >>= 2 is same as C = C >> 2
&=	Bitwise AND assignment operator	C &= 2 is same as C = C & 2
^=	bitwise exclusive OR and assignment operator	C ^= 2 is same as C = C ^ 2
|=	bitwise inclusive OR and assignment operator	C |= 2 is same as C = C | 2

Short Notes on L-VALUE and R-VALUE:

x = 1; takes the value on the right (e.g. 1) and puts it in the memory referenced by x. Here x and 1 are known as L-VALUES and R-VALUES respectively L-values can be on either side of the assignment operator where as R-values only appear on the right.

So x is an L-value because it can appear on the left as we've just seen, or on the right like this: y = x; However, constants like 1 are R-values because 1 could appear on the right, but 1 = x; is invalid.

Misc Operators

There are few other operators supported by C Language.

Operator	Description	Example
sizeof()	Returns the size of an variable.	size of(a), where a is integer, will return 4.
&	Returns the address of an variable.	&a; will give actual address of the variable.
*	Pointer to a variable.	*a; will pointer to a variable.

? :	Conditional Expression	If Condition is true ? Then value X : Otherwise value Y

Operators Categories:

All the operators we have discussed above can be categorized into following categories:

- Postfix operators, which follow a single operand.
- Unary prefix operators, which precede a single operand.
- Binary operators, which take two operands and perform a variety of arithmetic and logical operations.
- The conditional operator (a ternary operator), which takes three operands and evaluates either the second or third expression, depending on the evaluation of the first expression.
- Assignment operators, which assign a value to a variable.
- The comma operator, which guarantees left-to-right evaluation of comma-separated expressions.

Precedence of C Operators:

Operator precedence determines the grouping of terms in an expression. This affects how an expression is evaluated. Certain operators have higher precedence than others; for example, the multiplication operator has higher precedence than the addition operator:

For example x = 7 + 3 * 2; Here x is assigned 13, not 20 because operator * has higher precedenace than + so it first get multiplied with 3*2 and then adds into 7.

Here operators with the highest precedence appear at the top of the table, those with the lowest appear at the bottom. Within an expression, higher precedence operators will be evaluated first.

Category	**Operator**	**Associativity**
Postfix	() [] -> . ++ - -	Left to right
Unary	+ - ! ~ ++ - - (type) * & sizeof	Right to left
Multiplicative	* / %	Left to right
Additive	+ -	Left to right
Shift	<< >>	Left to right
Relational	< <= > >=	Left to right
Equality	== !=	Left to right

Bitwise AND	&	Left to right
Bitwise XOR	^	Left to right
Bitwise OR	|	Left to right
Logical AND	&&	Left to right
Logical OR	||	Left to right
Conditional	?:	Right to left
Assignment	= += -= *= /= %= >>= <<= &= ^= |=	Right to left
Comma	,	Left to right

Operator and Expression

The C programming language provides a rich set of operators that you can use in manipulating data.

Operators are symbols like "**+**" and "**=**" that are used in conjunction with variables and constants in order to create expressions. Expressions are C statements that which when evaluated, will produce data.

I'll group the operators according to their use below. Now, even though we'll discuss assignment operators in detail further down below, please note that the basic assignment operator is the equal sign "**=**". This is slightly different from the equality symbol in Mathematics.

Therefore in the sample code below, we are not testing whether **var** is equal to **10** but we are assigning the value **10** to the variable **var**.

?

```
var = 10;
```

Arithmetic operators

We'll look at first the so-called "arithmetic" operators which you may be well familiar with.

Symbol	Meaning
*	multiplication
/	division
%	modulo
+	addition
-	subtraction

The operators above are also considered as "**binary**" operators. Please don't think that they're used in manipulating binary data. The term binary simply means that these operators need **two** values in order to work properly. See examples below:

?

```
x = 2 + 3;
x = 10 * 5;
x = 18 - 5;
x = 5 / 4;
```

Note from the examples above that there is a value on the left and on the right of the arithmetic operators used. These are actually called "**operands**". Operands can also be variables. Furthermore, you can even put multiple operators and operands in one line of code. See below for more examples.

?

```
x = 2 + 3 * 5;
y = x * x;
z = 120 * 5 - x;
```

Now you might be thinking, if there are multiple operators in one line, what is the order in which they are evaluated? Luckily C follows the convention in Mathematics. To recall, here's the order of operations to be taken, this is normally known as the **precedence rule**.

1. Expressions inside parentheses will be evaluated first
2. then Terms with exponents
3. then Multiplication, division and modulo
4. and finally, Addition and subtraction

That's pretty straightforward isn't it. I'll leave it to you to experiment with this.

For those who don't know yet, **modulo** is used to get the remainder of a division operation. Like **10 % 3** will give you **1**. That's because **10 / 3** = **3 remainder 1**.

Relational and Logical operators

The relational and logical operators are used for comparing two values. These are again considered as binary operators requiring two operands in order to function properly.

We will get into discussing **conditional statements** in C later on as part of the topic "**Controlling program flow**" where we'll discuss the keywords **if**, **switch**and loops.

For now, just familiarize yourself with the operators and their meaning below. Also note that relational and logical operators are lower in precedence than arithmetic operators. In order words, given:

x < y + 5

the compiler will interpret this as

x < (y + 5)

In other words, **y + 5** will be evaluated first and then compare the result with **x**.

Below is a table showing the relational and logical operators arranged in order of precedence. Symbols at the top has higher precedence than symbols at the bottom.

Symbol	Meaning
<	less than
<=	less than or equal
>	greater than
>=	greater than or equal

Symbol	Meaning
==	is equal
!=	is not equal
&&	logical AND
||	logical OR

Bitwise operators

Bitwise operators are used to perform bit manipulations. There are six such operators used in C and are shown in the table below.

Symbol	Meaning
<<	left shift
>>	right shift
&	bitwise AND
|	bitwise OR
^	bitwise XOR
~	bitwise NOT

As a short review of bitwise operations, let's recall that data at the lowest level is represented by the computer in binary digits or "bits". Data can only be represented using digits 1 and 0. For instance, the decimal value **65** will be stored in the computer as **01000001**. This can also be represented in hexadecimal ("base 16") notation as **0x41**.

Therefore bitwise operations refer to operations that affect the value of a bit or bits that represent data. For example, the expression below will have the effect of converting the decimal value 65 into decimal value 130. Or in hexadecimal, 0x41 is converted into 0x82.

?

```
65 << 1
```

So what's happened here? So given the decimal 65, which is represented in binary as:

0 1 0 0 0 0 0 1

Note that I've only shown the lower 8 bits of the number. In reality, the number will be stored in the computer using either 32, 16 or 8 bits depending on the data type. When the left shift is applied to it using the "<<" operator. All the bits are moved to the left by one location. Since there are only so many bits that can hold the number, doing the left shift would have the effect of making the left-most bit to fall off and the right most bit location to be empty.

So if, for instance, the value 65 above is of type char, then there are only 8 bit locations available to store it. Thus with the left shift of 1 bit location, here's what will happen.

$_{\mathbf{0}}$ **1 0 0 0 0 0 1** $\underline{\mathbf{0}}$

The left-most zero will be dropped, then the next bit, which is 1, will take its place and will become the new left-most bit. Then the next zero and all the other zeros after that will move to the left in the same manner.

Because the bits move in that fashion, a new blank bit location will appear at the right-most bit. This empty location will be taken by a value of zero. It's like as if this new zero on the right has pushed all the bits to the left dropping that left-most zero off in the process.

What you're left with is:

1 0 0 0 0 0 1 0

Which is 130 in decimal and 0x82 in hexadecimal.

Assignment operators

Assignment operators are operators used to assign the value of an expression in a variable or constant. We've seen that equal sign operator which as we know very well assigns the value on the right to the identifier on the left. There are however other operators you can use, like "**+=**", that's "plus" and "equal" signs together.

The "**+**" symbol can be any of the above binary operators like -, <<, *, etc. This type of assignment operator is used usually to increment or decrement a variable. Let's say you have an expression like this:

x = x + 5;

We know you are assigning **x + 5** to the variable **x**. This can be written as:

x += 5;

The expression above would usually be read as "increment **x** by **5**". Again, this applies to the other binary operators above including the bitwise operators. So you can have the following examples:

?

```
x *= 100;
y <<= 1;
z &= 0xff;
```

Unary operators

Unary operators are operators that require only one operand. This will include the following:

Symbol	Meaning
!	logical NOT
~	bitwise NOT
+, -	positive, negative
++	increment by one
--	decrement by one
*	dereference
&	address of

The logical NOT will make a non-zero value zero and will make a zero value 1. For example:

!5 means **0**

!0 means **1**

The bitwise NOT will reverse the bits of a value. So for example:

~7 means **-8**

This bitwise NOT operation can be tricky and will require you to review your one's and two's complement to better understand them.

The increment and decrement operators are similar to += and -= assignment operators above. This time though, they always add or subtract one from the variable.

The * and & operators here, which can be confused with the multiplication and bitwise AND operators, are used in an advanced variable called the **pointer**. We'll have fun with pointers later on but not now.

All this might be quite a lot to digest for now, so I'll stop here and move on to other more exciting topics next time. As an exercise for you, what would happen if the result of an expression does not match the target variable ("the variable on the left side of the equation"). Like the example below where by **x** is an integer but **5 / 4** will yield **1.25** which is a float?

int x = 5 / 4;

Or what about this next one, where by the result will be **36100** which is more than what can be stored in a **char** type variable?

char y = 190;

char z;

z = y * y;

Basics of Formatted Input/output in C

Concepts

- I/O is essentially done one character (or byte) at a time
- **stream** -- a sequence of characters flowing from one place to another
  - *input stream*: data flows from input device (keyboard, file, etc) into memory
  - *output stream*: data flows from memory to output device (monitor, file, printer, etc)
- **Standard I/O streams** (with built-in meaning)
  - `stdin`: standard input stream (default is keyboard)
  - `stdout`: standard output stream (defaults to monitor)
  - `stderr`: standard error stream
- `stdio.h` -- contains basic I/O functions
  - `scanf`: reads from standard input (`stdin`)
  - `printf`: writes to standard output (`stdout`)

- There are other functions similar to `printf` and `scanf` that write to and read from other streams
- How to include, for C or C++ compiler
- `#include <stdio.h>         // for a C compiler`
- `#include <cstdio>          // for a C++ compiler`

- *Formatted I/O* -- refers to the conversion of data to and from a stream of characters, for printing (or reading) in plain text format
    - All text I/O we do is considered *formatted* I/O
    - The other option is reading/writing direct binary information (common with file I/O, for example)

Output with `printf`

Recap

- The basic format of a **printf** function call is:
- `printf (`*`format_string`*`, `*`list_of_expressions`*`);`

  where:

    - *format_string* is the layout of what's being printed
    - *list_of_expressions* is a comma-separated list of variables or expressions yielding results to be inserted into the output

- To output string literals, just use one parameter on `printf`, the string itself
- `printf("Hello, world!\n");`
- `printf("Greetings, Earthling\n\n");`

Unformatted I/O functions...(such as getchar, fgets)
Formatted I/O functions.... (such as scanf, fscanf)

But,,, cant quite put my fingure on the differences between unformatted I/O and formatted I/O.

printf is formatted output. Is it because whatever you write between the
" " can be printed out (output) to the screen and that is refered as "being formatted"??

getchar is UNformatted input. Is it because it takes only one char each? (or did it take string also..?)

formatted I/O would probably be those functions accepting a format string: that is the first parameter in printf/scanf etc:
:
: printf("%s %d I am the format string", str, i);

For example, looking the prototype of **putc(int c, FILE* stream)**.
This is to **Writes one btye to *stream*. I thought the stream is the string but not? It could be a character one by one? (the stream composed by characters, one by one, but NOT as a string)**

Another example, gets(char* s). This is to Reads one line from standard input. Removed the ending newline character.
This is also a function for unformatted I/O but, it takes string. I thought strings were formatted I/O..?

Another example is reformatting a string:
```
: : :
: : : char* a="18";
: : : printf("32%s97", a);
: : :
```
Formatted I/O means that presenting data in another form than it exists with.
: For example, in memory I have the integer 321897, which is stored in four bytes (on a 32-bit machine) in binary.
: Presenting this value to the user, on-screen for example, requires reformatting the four bytes to a readable 6-digit string:
```
:
: int a=321897
: printf("%d", a);
:

:
```
: Another example is reformatting a string:
```
:
: char* a="18";
: printf("32%s97", a);
```

Decision Making and Looping using while statement, do while loop , for statement week No: 17/18

While Statement
while (condition) { body of loop }

```
i=0;

while (i<10)

{       printf (“%f”, i*i);

i++;        }

ans = ‘y’;

while (ans==‘y’)    {/*process*/

printf (“Do you want to continue y/n”)

scanf(“%c”, ans); }
```

Do While Loop
do {body of the loop} while (condition);

It’s exit controlled loop

```
i=0;

do

{ printf (“%f”, i*i); i++; }
```

while (i<10);

while (number > 0 && number < 12)

The de Morgan's law

For Statement
for(initialization; condition; increment)

for(i=0;i<10;++i) {printf (“%d”,i);}

for(i=10; i>0; --i) {printf (%d”, i);}

for(i=0,j=0;i<10;i++)

for(i=0,j =0;i<10;i++,j++)

for(i=0,j=0; i<50 || j <10;j++)

for (i=0;i<10000;i++)   ;

Execution of For Loop

- Initialization of control variable is done first
- The value of control variable is tested then
- If the test condition value is true, the loop executes, otherwise the body gets skipped
- After the first execution of the loop the control goes to the third part of for
    - for (total=0;ans=“n”; total = total + marks) {scanf (“%f”, marks; /* ask for yes no*/}

Different Ways to Use For
i = 5;

for (; i < 10 ; i++)

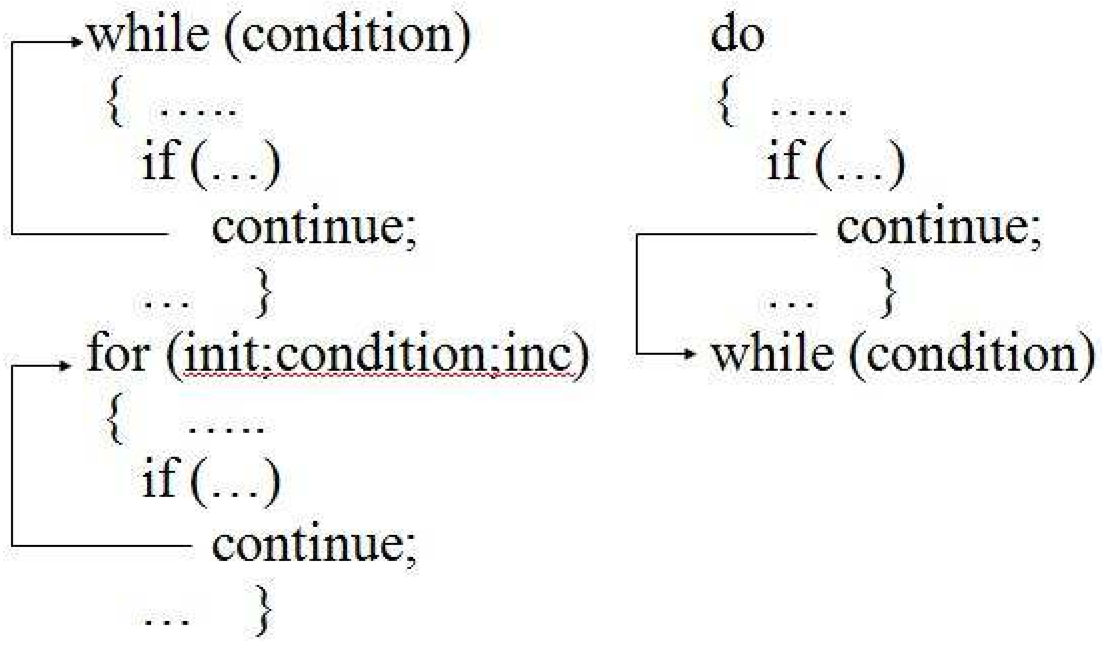
i = 5;

```
for (; i<10; ) { … ; i++; …}

for (; ;) { … } /* The infinite loop */

for (row =0; row < rowmax; ++row)

{ for (col = 0; col < colmax; ++ col;)

printf( "%d", row * col;);  }

/* Above is known as nesting */
```

Jumps in Loops

- break and goto for jumping out
- goto for jumping anywhere from anywhere!
- Skipping next bunch of statements and continue with next iteration by continue
- continue does not breaks out of loop
- In while and do loops, continue causes control to go to condition, in for it goes to increment section and then to condition

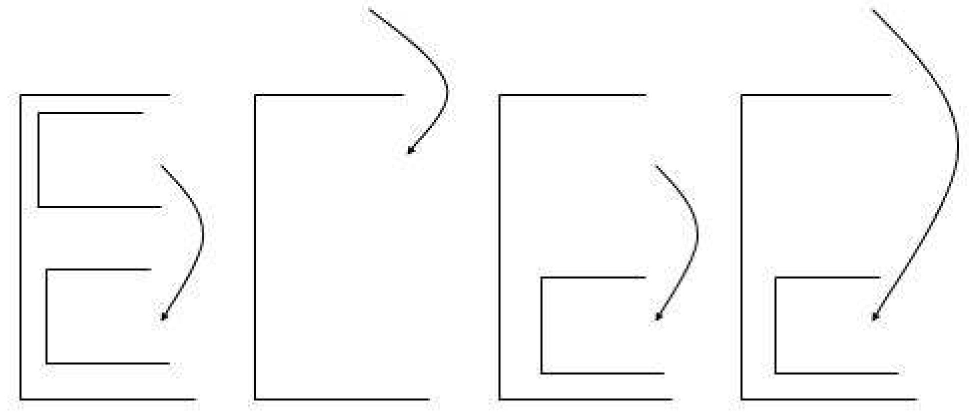
Continue Graphically



Avoiding goto

- It generates less efficient code
- Careful program design can usually avoid use of goto
- Many of goto will make the program logic complicated and less readable
- In case any goto is absolutely necessary, it should be properly documented

goto Which Causes Problems



Concise Test Expressions
if (expression = 0) can be rewritten as

if (!expression)

if (expression != 0) can be rewritten as

if (expression)

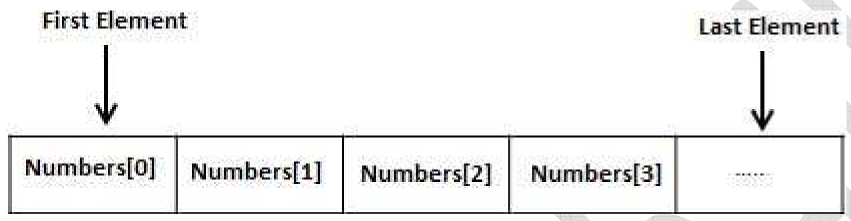
- The second form is more common in professional C programs
- It does not use any relational operators

C - Arrays

C programming language provides a data structure called **the array**, which can store a fixed-size sequential collection of elements of the same type. An array is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same type.

Instead of declaring individual variables, such as number0, number1, ..., and number99, you declare one array variable such as numbers and use numbers[0], numbers[1], and ..., numbers[99] to represent individual variables. A specific element in an array is accessed by an index.

All arrays consist of contiguous memory locations. The lowest address corresponds to the first element and the highest address to the last element.



Declaring Arrays

To declare an array in C, a programmer specifies the type of the elements and the number of elements required by an array as follows:

```
type arrayName [ arraySize ];
```

This is called a *single-dimensional* array. The **arraySize** must be an integer constant greater than zero and **type** can be any valid C data type. For example, to declare a 10-element array called **balance** of type double, use this statement:

```
double balance[10];
```

Now *balance* is avariable array which is sufficient to hold upto 10 double numbers.

Initializing Arrays

You can initialize array in C either one by one or using a single statement as follows:

```
double balance[5] = {1000.0, 2.0, 3.4, 17.0, 50.0};
```

The number of values between braces { } can not be larger than the number of elements that we declare for the array between square brackets [ ]. Following is an example to assign a single element of the array:

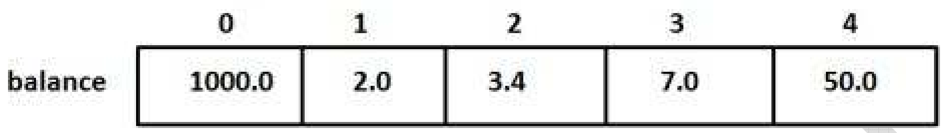
If you omit the size of the array, an array just big enough to hold the initialization is created. Therefore, if you write:

```
double balance[] = {1000.0, 2.0, 3.4, 17.0, 50.0};
```

You will create exactly the same array as you did in the previous example.

```
balance[4] = 50.0;
```

The above statement assigns element number 5th in the array a value of 50.0. Array with 4th index will be 5th ie. last element because all arrays have 0 as the index of their first element which is also called base index. Following is the pictorial representation of the same array we discussed above:



Accessing Array Elements

An element is accessed by indexing the array name. This is done by placing the index of the element within square brackets after the name of the array. For example:

```
double salary = balance[9];
```

The above statement will take 10th element from the array and assign the value to salary variable. Following is an example which will use all the above mentioned three concepts viz. declaration, assignment and accessing arrays:

```
#include <stdio.h>

int main ()
{
int n[ 10 ]; /* n is an array of 10 integers */
int i,j;

/* initialize elements of array n to 0 */
for ( i = 0; i < 10; i++ )
{
n[ i ] = i + 100; /* set element at location i to i + 100 */
}

/* output each array element's value */
for (j = 0; j < 10; j++ )
{
printf("Element[%d] = %d\n", j, n[j] );
}

return 0;
}
```

When the above code is compiled and executed, it produces following result:

```
Element[0] = 100
Element[1] = 101
Element[2] = 102
Element[3] = 103
Element[4] = 104
Element[5] = 105
Element[6] = 106
```

```
Element[7] = 107
Element[8] = 108
Element[9] = 109
```

C Arrays in Detail

Arrays are important to C and should need lots of more detail. There are following few important concepts related to array which should be clear to a C programmer:

Concept	Description
Multi-dimensional arrays	C supports multidimensional arrays. The simplest form of the multidimensional array is the two-dimensional array.
Passing arrays to functions	You can pass to the function a pointer to an array by specifying the array's name without an index.
Return array from a function	C allows a function to return an array.
Pointer to an array	You can generate a pointer to the first element of an array by simply specifying the array name, without any index.

C string handling

C string handling refers to a group of functions implementing operations on strings in the C standard library. Various operations, such as copying, concatenation, tokenization and searching are supported.

The only support for strings in the C programming language itself is that the compiler will translate a quoted string constant into a null-terminated string, which is stored in static memory. However, the standard C library provides a large number of functions designed to manipulate these null-terminated strings. These functions are so popular and used so often that they are usually considered part of the definition of C.

Definitions

A string is a contiguous sequence of characters terminated by and including the first null character (written `'\0'` and corresponding to the ASCII character NUL). In C, there are two types of strings: **string**, which is sometimes called **byte string**, and **wide string**.[1] A byte string contains the type `char`s as code units (one `char` is at least 8 bits), whereas a wide string contains the type `wchar_t` as code units.

A common misconception is that all `char` arrays are strings, because string literals are converted to arrays during the compilation (or translation) phase.[2] It is important to remember that a string **ends** at the first null character. An array or string literal that contains a null character before the last byte therefore *contains* a string, or possibly several strings, but is not itself a string.[3] Conversely, it is possible to create a `char` array that is not null-terminated and is thus not a string: `char` is often used as integer when needing to save memory, for example when having an array of booleans.

The term **pointer to a string** is used in C to describe a pointer to the initial (lowest-addressed) byte of a string.[1] In C, pointers are used to pass strings to functions. Documentation (including this page) will often use the term *string* to mean*pointer to a string*.

The term **length of a string** is used in C to describe the number of bytes preceding the null character.[1] `strlen` is a standardised function commonly used to determine the length of a string.

Character encodings

Each string ends at the first occurrence of the null character of the appropriate kind (`char` or `wchar_t`). A null character is a character represented as a zero. Consequently, a byte string can contain non-NUL characters in ASCII or any ASCII extension, but not characters in encodings such as UTF-16 (even though a 16-bit code unit might be nonzero, its high or low byte might be zero). The encodings that can be stored in wide strings are defined by the width of `wchar_t`. In most implementations,`wchar_t` is at least 16 bits, and so all 16-bit encodings, such as UCS-2, can be stored. If `wchar_t` is 32-bits, then 32-bit encodings, such as UTF-32, can be stored.

Variable-width encodings can be used in both byte strings and wide strings. String length and offsets are measured in bytes or`wchar_t`, not in "characters", which can be confusing to beginning programmers. UTF-8 and Shift JIS are often used in C byte strings, while UTF-16 is often used in C wide strings when `wchar_t` is 16 bits. Truncating strings with variable length characters using functions like `strncpy` can produce invalid sequences at the end of the string. This can be unsafe if the truncated parts are interpreted by code that assumes the input is valid.

Support for Unicode literals such as `char foo[512] = "φωωβαρ";`(UTF-8) or `wchar_t foo[512] = L"φωωβαρ";` (UTF-16 or UTF-32) is implementation defined,[4] and may require that the source code be in the same encoding. Some compilers or editors will require entering all non-ASCII characters as `\xNN` sequences for each byte of UTF-8, and/or `\uNNNN` for each word of UTF-16.

Constants and types

Name	Notes

`NULL`	macro expanding to the null pointer constant; that is, a constant representing a pointer value which is guaranteed **not** to be a valid address of an object in memory.
`wchar_t`	type used for a code unit in a wide strings, usually either 16 or 32 bits.
`wint_t`	integer type that can hold any value of a wchar_t as well as the value of the macro WEOF. This type is unchanged by integral promotions. Usually a 32 bit signed value.
`mbstate_t`	contains all the information about the conversion state required from one call to a function to the other.

Functions

	Byte string	**Wide string**	**Description**[note 1]
String manipulation	`strcpy`	`wcscpy`	copies one string to another
	`strncpy`	`wcsncpy`	writes exactly n bytes/`wchar_t`, copying from source or adding nulls
	`strcat`	`wcscat`	appends one string to another
	`strncat`	`wcsncat`	appends no more than n bytes/`wchar_t` from one string to another
	`strxfrm`	`wcsxfrm`	transforms a string according to the current locale
String	`strlen`	`wcslen`	returns the length of the string

examination	strcmp	wcscmp	compares two strings
	strncmp	wcsncmp	compares a specific number of bytes/wchar_t in two strings
	strcoll	wcscoll	compares two strings according to the current locale
	strchr	wcschr	finds the first occurrence of a byte/wchar_t in a string
	strrchr	wcsrchr	finds the last occurrence of a byte/wchar_t in a string
	strspn	wcsspn	finds in a string the first occurrence of a byte/wchar_t not in a set
	strcspn	wcscspn	finds in a string the last occurrence of a byte/wchar_t not in a set
	strpbrk	wcspbrk	finds in a string the first occurrence of a byte/wchar_t in a set
	strstr	wcsstr	finds the first occurrence of a substring in a string
	strtok	wcstok	splits string into tokens
Miscellaneous	strerror	N/A	returns a string containing a message derived from an error code
Memory manipulation	memset	wmemset	fills a buffer with a repeated byte/wchar_t
	memcpy	wmemcpy	copies one buffer to another

	memmove	wmemmove	copies one buffer to another, possibly overlapping, buffer
	memcmp	wmemcmp	compares two buffers
	memchr	wmemchr	finds the first occurrence of a byte/`wchar_t` in a buffer

1. ^ Here *string* refers either to byte string or wide string

Multibyte functions

Name	Description
mblen	returns the number of bytes in the next multibyte character
mbtowc	converts the next multibyte character to a wide character
wctomb	converts a wide character to its multibyte representation
mbstowcs	converts a multibyte string to a wide string
wcstombs	converts a wide string to a multibyte string
btowc	convert a single-byte character to wide character, if possible
wctob	convert a wide character to a single-byte character, if possible
mbsinit	checks if a state object represents initial state
mbrlen	returns the number of bytes in the next multibyte character, given state

mbrtowc	converts the next multibyte character to a wide character, given state
wcrtomb	converts a wide character to its multibyte representation, given state
mbsrtowcs	converts a multibyte string to a wide string, given state
wcsrtombs	converts a wide string to a multibyte string, given state

"state" is used by encodings that rely on history such as shift states. This is not needed by UTF-8 or UTF-32. UTF-16 uses them to keep track of surrogate pairs and to hide the fact that it actually is a multi-word encoding.

Numeric conversions

The C standard library contains several functions for numeric conversions. The functions that deal with byte strings are defined in the `stdlib.h` header (`cstdlib` header in C++). The functions that deal with wide strings are defined in the`wchar.h` header (`cwchar` header in C++). Note that the `strtoxxx` functions are not const-correct, since they accept a `const`string pointer and return a non-`const` pointer within the string.

Byte string	Wide string	Description[note 1]
atof	N/A	converts a string to a floating-point value
atoi atol atoll	N/A	converts a string to an integer (C99)

strtof(C99)
strtod
strtold(C99)
	wcstof(C99)
	wcstod
	wcstold(C99)
		converts a string to a floating-point value

strtol strtoll	wcstol wcstoll	converts a string to a signed integer
strtoul strtoull	wcstoul wcstoull	converts a string to an unsigned integer

C Programming User-defined functions Week No:19/21

This chapter is the continuation to the function Introduction chapter.

Example of user-defined function

Write a C program to add two integers. Make a function add to add integers and display sum in main() function.

```
/*Program to demonstrate the working of user defined function*/
#include <stdio.h>
int add(int a, int b);          //function prototype(declaration)
int main(){
int num1,num2,sum;
printf("Enters two number to add\n");
scanf("%d %d",&num1,&num2);
sum=add(num1,num2);          //function call
printf("sum=%d",sum);
return 0;
}
int add(int a,int b)            //function declarator
{
/* Start of function definition. */
int add;
add=a+b;
return add;                     //return statement of function
/* End of function definition. */
}
```

Function prototype(declaration):

Every function in C programming should be declared before they are used. These type of declaration are also called function prototype. Function prototype gives compiler information about function name, type of arguments to be passed and return type.

Syntax of function prototype

```
return_type function_name(type(1) argument(1),....,type(n)
argument(n));
```

In the above example,int add(int a, int b); is a function prototype which provides following information to the compiler:

1. name of the function is add()
2. return type of the function is int.
3. two arguments of type int are passed to function.

Function prototype are not needed if user-definition function is written before main() function.

Function call

Control of the program cannot be transferred to user-defined function unless it is called invoked).

Syntax of function call

```
Function name (argument (1),....argument(n));
```

In the above example, function call is made using statement add(num1,num2); from main(). This make the control of program jump from that statement to function definition and executes the codes inside that function.

Function definition

Function definition contains programming codes to perform specific task.

Syntax of function definition

```
return_type function_name(type(1) argument(1),..,type(n) argument(n))
{
//body of function
}
```

Function definition has two major components:

1. Function declarator

Function declarator is the first line of function definition. When a function is invoked from calling function, control of the program is transferred to function declarator or called function.

Syntax of function declarator

```
return_type function_name(type(1) argument(1),....,type(n)
argument(n))
```

Syntax of function declaration and declarator are almost same except, there is no semicolon at the end of declarator and function declarator is followed by function body.

In above example, int add(int a,int b) in line 12 is a function declarator.

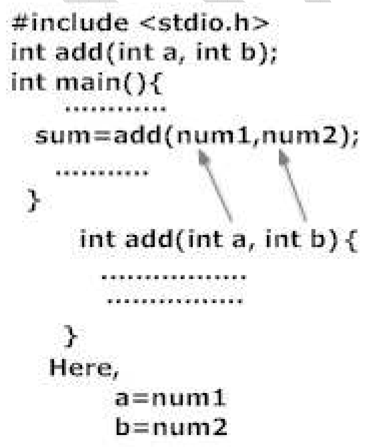
2. Function body

Function declarator is followed by body of function which is composed of statements.

Passing arguments to functions

In programming, argument/parameter is a piece of data(constant or variable) passed from a program to the function.

In above example two variable, num1 and num2 are passed to function during function call and these arguments are accepted by arguments a and b in function definition.



```
#include <stdio.h>
int add(int a, int b);
int main(){
    ............
  sum=add(num1,num2);
   ...........
 }
      int add(int a, int b){
        ................
        ................
     }
   Here,
        a=num1
        b=num2
```

Arguments that are passed in function call and arguments that are accepted in function definition should have same data type. For example:

If argument num1 was of int type and num2 was of float type then, argument variable a should be of type int and b should be of type float,i.e., type of argument during function call and function definition should be same.

A function can be called with or without an argument.

Return Statement

Return statement is used for returning a value from function definition to calling function.

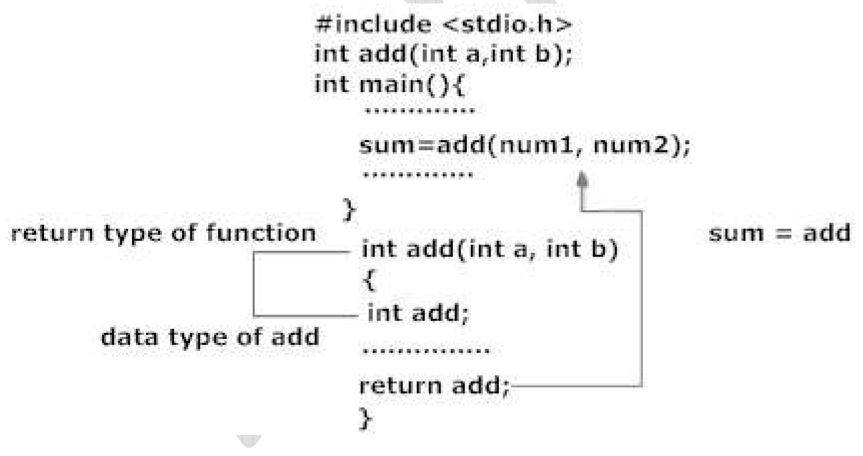
Syntax of return statement

```
return (expression);
OR
return;
```

For example:

```
return;
return a;
return (a+b);
```

In above example, value of variable add in add() function is returned and that value is stored in variable sum in main() function. The data type of expression in return statement should also match the return type of function.



```
#include <stdio.h>
int add(int a,int b);
int main(){
   .............
   sum=add(num1, num2);
   .............
  }
   int add(int a, int b)
   {
    int add;
   ...............
   return add;
   }
```

C Programming Pointers Week no: 22/23

Pointers are the powerful feature of C and (C++) programming, which differs it from other popular programming languages like: java and Visual Basic.

Pointers are used in C program to access the memory and manipulate the address.

Reference operator(&)

If var is a variable then, &var is the address in memory.

```
/* Example to demonstrate use of reference operator in C programming.
*/
#include <stdio.h>
int main(){
int var=5;
printf("Value: %d\n",var);
printf("Address: %d",&var);  //Notice, the ampersand(&) before var.
return 0;
}
```

Output

```
Value: 5
Address: 2686778
```

Note: You may obtain different value of address while using this code.

In above source code, value 5 is stored in the memory location 2686778. var is just the name given to that location.

You, have already used reference operator in C program while using scanf() function.

```
scanf("%d",&var);
```

Reference operator(*) and Pointer variables

Pointers variables are used for taking addresses as values, i.e., a variable that holds address value is called a pointer variable or simply a pointer.

Declaration of Pointer

Dereference operator(*) are used to identify an operator as a pointer.

```
data_type * pointer_variable_name;
int *p;
```

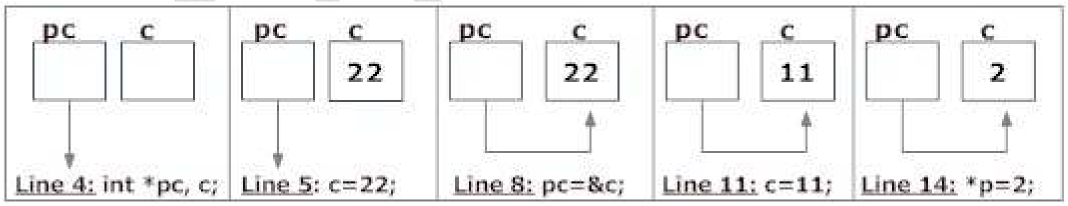
Above statement defines, p as pointer variable of type int.

Example To Demonstrate Working of Pointers

```c
/* Source code to demonstrate, handling of pointers in C program */
#include <stdio.h>
int main(){
int *pc,c;
c=22;
printf("Address of c:%d\n",&c);
printf("Value of c:%d\n\n",c);
pc=&c;
printf("Address of pointer pc:%d\n",pc);
printf("Content of pointer pc:%d\n\n",*pc);
c=11;
printf("Address of pointer pc:%d\n",pc);
printf("Content of pointer pc:%d\n\n",*pc);
*pc=2;
printf("Address of c:%d\n",&c);
printf("Value of c:%d\n\n",c);
return 0;
}
```

Output

```
Address of c: 2686784
Value of c: 22

Address of pointer pc: 2686784
Content of pointer pc: 22

Address of pointer pc: 2686784
Content of pointer pc: 11

Address of c: 2686784
Value of c: 2
```



Explanation of program and figure

1. Code int *pc, p; creates a pointer pc and a variable c. Pointer pc points to some address and that address has garbage value. Similarly, variable c also has garbage value at this point.
2. Code c=22; makes the value of c equal to 22, i.e.,22 is stored in the memory location of variable c.
3. Code pc=&c; makes pointer, point to address of c. Note that, &c is the address of variable c (because c is normal variable) and pc is the address of pc (because pc is the pointer variable). Since the address of pc and address of c is same, *pc (value of pointer pc) will be equal to the value of c.
4. Code c=11; makes the value of c, 11. Since, pointer pc is pointing to address of c. Value of *pc will also be 11.
5. Code *pc=2; change the address pointed by pointer pc to change to 2. Since, address of pointer pc is same as address of c, value of c also changes to 2.

Commonly done mistakes in pointers

Suppose, the programmar want pointer pc to point to the address of c. Then,

```
int c, *pc;
pc=c;  /* pc is address whereas, c is not an address. */
*pc=&c; /* &c is address whereas, *pc is not an address. */
```

In both cases, pointer pc is not pointing to the address of c.

File management in C

Author: Ramesh B   Published on: 31st May 2006 | Last Updated on: 4th Mar 2011

C Programming - File management in C

In this tutorial you will learn about C Programming - File management in C, File operation functions in C, Defining and opening a file, Closing a file, The getw and putw functions, The fprintf & fscanf functions, Random access to files and fseek function.

C supports a number of functions that have the ability to perform basic file operations, which include:
1. Naming a file
2. Opening a file
3. Reading from a file
4. Writing data into a file
5. Closing a file

- Real life situations involve large volume of data and in such cases, the console oriented I/O operations pose two major problems
- It becomes cumbersome and time consuming to handle large volumes of data through terminals.
- The entire data is lost when either the program is terminated or computer is turned off therefore it is necessary to have more flexible approach where data can be stored on the disks and read whenever necessary, without destroying the data. This method employs the concept of files to store data.

File operation functions in C:

Function Name	Operation
fopen()	Creates a new file for use Opens a new existing file for use
fclose	Closes a file which has been opened for use
getc()	Reads a character from a file
putc()	Writes a character to a file
fprintf()	Writes a set of data values to a file

fscanf()	Reads a set of data values from a file
getw()	Reads a integer from a file
putw()	Writes an integer to the file
fseek()	Sets the position to a desired point in the file
ftell()	Gives the current position in the file
rewind()	Sets the position to the begining of the file

Defining and opening a file:

If we want to store data in a file into the secondary memory, we must specify certain things about the file to the operating system. They include the fielname, data structure, purpose.

The general format of the function used for opening a file is

```
FILE *fp;
fp=fopen("filename","mode");
```

The first statement declares the variable fp as a pointer to the data type FILE. As stated earlier, File is a structure that is defined in the I/O Library. The second statement opens the file named filename and assigns an identifier to the FILE type pointer fp. This pointer, which contains all the information about the file, is subsequently used as a communication link between the system and the program. The second statement also specifies the purpose of opening the file. The mode does this job.

R open the file for read only.
W open the file for writing only.
A open the file for appending data to it.

Consider the following statements:

```
FILE *p1, *p2;
p1=fopen("data","r");
p2=fopen("results","w");
```

In these statements the p1 and p2 are created and assigned to open the files data and results respectively the file data is opened for reading and result is opened for writing.

In case the results file already exists, its contents are deleted and the files are opened as a new file. If data file does not exist error will occur.

Closing a file:

The input output library supports the function to close a file; it is in the following format.
fclose(file_pointer);

A file must be closed as soon as all operations on it have been completed. This would close the file associated with the file pointer.
Observe the following program.

```
….
FILE *p1 *p2;
p1=fopen (“Input”,”w”);
p2=fopen (“Output”,”r”);
….
…
fclose(p1);
fclose(p2)
```

The above program opens two files and closes them after all operations on them are completed, once a file is closed its file pointer can be reversed on other file.

The getc and putc functions are analogous to getchar and putchar functions and handle one character at a time. The putc function writes the character contained in character variable c to the file associated with the pointer fp1. ex putc(c,fp1); similarly getc function is used to read a character from a file that has been open in read mode. c=getc(fp2).

The program shown below displays use of a file operations. The data enter through the keyboard and the program writes it. Character by character, to the file input. The end of the data is indicated by entering an EOF character, which is control-z. the file input is closed at this signal.

```
#include< stdio.h >
main()
{
file *f1;
printf(“Data input output”);
f1=fopen(“Input”,”w”); /*Open the file Input*/
while((c=getchar())!=EOF) /*get a character from key
board*/
putc(c,f1); /*write a character to input*/
fclose(f1); /*close the file input*/
printf("nData outputn");
f1=fopen("INPUT","r"); /*Reopen the file input*/
while((c=getc(f1))!=EOF)
printf("%c",c);
fclose(f1);
}
```

The getw and putw functions:

These are integer-oriented functions. They are similar to get c and putc functions and are used to read and write integer values. These functions would be usefull when we deal with only integer data. The general forms of getw and putw are:

```
putw(integer,fp);
getw(fp);

/*Example program for using getw and putw functions*/
#include< stdio.h >
main()
{
FILE *f1,*f2,*f3;
int number I;
printf("Contents of the data filenn");
f1=fopen("DATA","W");
for(I=1;I< 30;I++)
{
scanf("%d",&number);
if(number==-1)
break;
putw(number,f1);
}
fclose(f1);
f1=fopen("DATA","r");
f2=fopen("ODD","w");
f3=fopen("EVEN","w");
while((number=getw(f1))!=EOF)/* Read from data file*/
{
if(number%2==0)
putw(number,f3);/*Write to even file*/
else
putw(number,f2);/*write to odd file*/
}
fclose(f1);
fclose(f2);
fclose(f3);
f2=fopen("ODD","r");
f3=fopen("EVEN","r");
printf("nnContents of the odd filenn");
while(number=getw(f2))!=EOF)
printf("%d%d",number);
printf("nnContents of the even file");
while(number=getw(f3))!=EOF)
printf("%d",number);
fclose(f2);
fclose(f3);
}
```

The fprintf & fscanf functions:

The fprintf and fscanf functions are identical to printf and scanf functions except that they work on files. The first argument of theses functions is a file pointer which specifies the file to be used. The general form of fprintf is

```
fprintf(fp,"control string", list);
```

Where fp id a file pointer associated with a file that has been opened for writing. The control string is file output specifications list may include variable, constant and string.

```
fprintf(f1,%s%d%f",name,age,7.5);
```

Here name is an array variable of type char and age is an int variable
The general format of fscanf is

```
fscanf(fp,"controlstring",list);
```

This statement would cause the reading of items in the control string.

Example:

```
fscanf(f2,"5s%d",item,&quantity");
```

Like scanf, fscanf also returns the number of items that are successfully read.

```
/*Program to handle mixed data types*/
#include< stdio.h >
main()
{
FILE *fp;
int num,qty,I;
float price,value;
char item[10],filename[10];
printf("Input filename");
scanf("%s",filename);
fp=fopen(filename,"w");
printf("Input inventory datann"0;
printf("Item namem number price quantityn");
for I=1;I< =3;I++)
{
fscanf(stdin,"%s%d%f%d",item,&number,&price,&quality);
fprintf(fp,"%s%d%f%d",itemnumber,price,quality);
}
fclose (fp);
fprintf(stdout,"nn");
fp=fopen(filename,"r");
printf("Item name number price quantity value");
for(I=1;I< =3;I++)
{
fscanf(fp,"%s%d%f%d",item,&number,&prince,&quality);
value=price*quantity");
fprintf("stdout,"%s%d%f%d%dn",item,number,price,quantity,
value);
}
fclose(fp);
}
```

Random access to files:

Sometimes it is required to access only a particular part of the and not the complete file. This can be accomplished by using the following function:

1 > fseek

fseek function:

The general format of fseek function is a s follows:

```
fseek(file pointer,offset, position);
```

This function is used to move the file position to a desired location within the file. Fileptr
is a pointer to the file concerned. Offset is a number or variable of type long, and position in an integer number. Offset specifies the number of positions (bytes) to be moved from the location specified bt the position. The position can take the 3 values.

Value	Meaning
0	Beginning of the file
1	Current position
2	End of the file.

First Sem. The End

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~The END ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~