

MDU ROHTAK

**MDU (ROHTAK)
B.TECH 1ST YEAR**

BY: NAEEM KHAN

SECTION A

**INTRODUCTION TO FUNDAMENTAL OF COMPUTER AND
PROGRAMMING**

Introduction

We know information processing, plays very important role in taking decision every moment. In this context, computers play a significant role in bulk of information processing. Here, we study what is a computer and organization of a computer. The computer operates on a program or set of instructions. We discuss the important contribution made by the John Von Neumann. The objective is to understand the definition of computer, working concepts of the computer, stored program concept and Microprocessor.

Definition of a Computer

A computer is a general purpose device which can be programmed to carry out a finite set of arithmetic or logical operations. Since a sequence of operations can be readily changed, the computer can solve more than one kind of problem. The essential point of a computer is to implement an idea, the terms of which are satisfied by Alan Turing's Universal Turing machine.

Conventionally, a computer consists of at least one processing element and some form of memory. The processing element carries out arithmetic and logic operations, and a sequencing and control unit that can change the order of operations based on stored information. Peripheral devices allow information to be retrieved from an external source, and the result of operations saved.

A computer's processing unit executes a series of instructions that make it read, manipulate and then store data. Conditional instructions change the sequence of instructions as a function of the current state of the machine or its environment.

Disadvantages:

1. It can have negative effects on your social life and interactions with other people if you do not maintain the balance between time online and offline.
2. It may have a negative effect on your eyesight due to radiation.
3. It may cause pimples and wrinkles.
4. It may distract you from your studies.
5. Too much time in front of monitor may adversely affect your eye sight.
6. Sitting in front of a computer for too long without exercise can cause a weight gain.

Advantages:

1. It helps you automate various tasks that you cannot do manually.
2. It helps you organize your data and information.
3. It has much more computing and calculating power than an ordinary human.
4. It may help your work to be a lot easier.
5. It can help you communicate with friends, coworkers and other contacts.

6. It has many search engines to help you find information quickly.

EVALUTION OF COMPUTERS

- The first counting device was the abacus, originally from Asia. It worked on a place-value notion meaning that the place of a bead or rock on the apparatus determined how much it was worth.
- **1600s:** John Napier discovers logarithms. Robert Bissaker invents the slide rule which will remain in popular use until 19??.
- **1642s:** Blaise Pascal, a French mathematician and philosopher, invents the first mechanical digital calculator using gears, called the Pascaline. Although this machine could perform addition and subtraction on whole numbers, it was too expensive and only Pascal himself could repair it.
- **1804s:** Joseph Marie Jacquard used punch cards to automate a weaving loom.
- **1812s:** Charles P. Babbage, the "father of the computer", discovered that many long calculations involved many similar, repeated operations. Therefore, he designed a machine, the difference engine which would be steam-powered, fully automatic and commanded by a fixed instruction program. In 1833, Babbage quit working on this machine to concentrate on the analytical engine.
- **1840s:** Augusta Ada. "The first programmer" suggested that a binary system should be used for storage rather than a decimal system.
- **1850s:** George Boole developed Boolean logic which would later be used in the design of computer circuitry.
- **1890s:** Dr. Herman Hollerith introduced the first electromechanical, punched-card data-processing machine which was used to compile information for the 1890 U.S. census. Hollerith's tabulator became so successful that he started his own business to market it. His company would eventually become International Business Machines (IBM).
- **1906s:** The vacuum tube is invented by American physicist Lee De Forest.
- **1939s:** Dr. John V. Atanasoff and his assistant Clifford Berry build the first electronic digital computer. Their machine, the Atanasoff-Berry-Computer (ABC) provided the foundation for the advances in electronic digital computers.
- **1941s:** Konrad Zuse (recently deceased in January of 1996), from Germany, introduced the first programmable computer designed to solve complex engineering equations. This machine, called the Z3, was also the first to work on the binary system instead of the decimal system.
- **1943s:** British mathematician Alan Turing developed a hypothetical device, the Turing machine which would be designed to perform logical operation and could read and write. It would presage

programmable computers. He also used vacuum technology to build British Colossus, a machine used to counteract the German code scrambling device, Enigma.

- **1944s:** Howard Aiken, in collaboration with engineers from IBM, constructed a large automatic digital sequence-controlled computer called the Harvard Mark I. This computer could handle all four arithmetic operations, and had special built-in programs for logarithms and trigonometric functions.
- **1945s:** Dr. John von Neumann presented a paper outlining the stored-program concept.
- **1947s:** The giant ENIAC (Electrical Numerical Integrator and Calculator) machine was developed by John W. Mauchly and J. Presper Eckert, Jr. at the University of Pennsylvania. It used 18, 000 vacuums, punch-card input, weighed thirty tons and occupied a thirty-by-fifty-foot space. It wasn't programmable but was productive from 1946 to 1955 and was used to compute artillery firing tables. That same year, the transistor was invented by William Shockley, John Bardeen and Walter Brattain of Bell Labs. It would rid computers of vacuum tubes and radios.
- **1949s:** Maurice V. Wilkes built the EDSAC (Electronic Delay Storage Automatic Computer), the first stored-program computer. EDVAC (Electronic Discrete Variable Automatic Computer), the second stored-program computer was built by Mauchly, Eckert and von Neumann. An Wang developed magnetic-core memory which Jay Forrester would reorganize to be more efficient.
- **1950s:** Turing built the ACE, considered by some to be the first programmable digital computer.

The First Generation (1951-1959)

- **1951s:** Mauchly and Eckert built the UNIVAC I, the first computer designed and sold commercially, specifically for business data-processing applications.
- **1950s :** Dr. Grace Murray Hopper developed the UNIVAC I compiler.
- **1957s:** The programming language FORTRAN (FORMula TRANslator) was designed by John Backus, an IBM engineer.
- **1959s:** Jack St. Clair Kilby and Robert Noyce of Texas Instruments manufactured the first integrated circuit, or chip, which is a collection of tiny little transistors.

The Second Generation (1959-1965)

- **1960s:** Gene Amdahl designed the IBM System/360 series of mainframe (G) computers, the first general-purpose digital computers to use integrated circuits.
- **1961s:** Dr. Hopper was instrumental in developing the COBOL (Common Business Oriented Language) programming language.

- **1963s:** Ken Olsen, founder of DEC, produced the PDP-I, the first minicomputer (G).
- **1965s:** BASIC (Beginners All-purpose Symbolic Instruction Code) programming language developed by Dr. Thomas Kurtz and Dr. John Kemeny.

The Third Generation (1965-1971)

- **1969s:** The Internet is started. (See History of the Internet)
- **1970s:** Dr. Ted Hoff developed the famous Intel 4004 microprocessor (G) chip.
- **1971s:** Intel released the first microprocessor, a specialized integrated circuit which was able to process four bits of data at a time. It also included its own arithmetic logic unit. PASCAL, a structured programming language, was developed by Niklaus Wirth.

The Fourth Generation (1971-1990)

- **1975s:** Ed Roberts, the "father of the microcomputer" designed the first microcomputer, the Altair 8800, which was produced by Micro Instrumentation and Telemetry Systems (MITS). The same year, two young hackers, William Gates and Paul Allen approached MITS and promised to deliver a BASIC compiler. So they did and from the sale, Microsoft was born.
- **1976s:** Cray developed the Cray-I supercomputer (G). Apple Computer, Inc was founded by Steven Jobs and Stephen Wozniak.
- **1977s:** Jobs and Wozniak designed and built the first Apple II microcomputer.
- **1970s: 1980:** IBM offers Bill Gates the opportunity to develop the operating system for its new IBM personal computer. Microsoft has achieved tremendous growth and success today due to the development of MS-DOS. Apple II was also released.
- **1981s:** The IBM PC was introduced with a 16-bit microprocessor.
- **1982s:** Time magazine chooses the computer instead of a person for its "Machine of the Year."
- **1984s:** Apple introduced the Macintosh computer, which incorporated a unique graphical interface, making it easy to use. The same year, IBM released the 286-AT.
- **1986s:** Compaq released the DeskPro 386 computer, the first to use the 80036 microprocessor.
- **1987s:** IBM announced the OS/2 operating-system technology.
- **1988s:** A nondestructive worm was introduced into the Internet network bringing thousands of computers to a halt.
- **1989s:** The Intel 486 became the world's first 1,000,000 transistor microprocessor.

The Fifth Generation (1990-present)

The 'Fifth Generation Computer Systems project (FGCS) was an initiative by Japan's Ministry of International Trade and Industry, begun in 1982, to create a "fifth generation computer" (see History of computing hardware) which was supposed to perform much calculation using massive parallel processing. It was to be the result of a massive government/industry research project in Japan during the 1980s. It aimed to create an "epoch-making computer" with supercomputer-like performance and to provide a platform for future developments in artificial intelligence.

The term fifth generation was intended to convey the system as being a leap beyond existing machines. Computers using vacuum tubes were called the first generation; transistors and diodes, the second; integrated circuits, the third; and those using microprocessors, the fourth. Whereas previous computer generations had focused on increasing the number of logic elements in a single CPU, the fifth generation, it was widely believed at the time, would instead turn to massive numbers of CPUs for added performance.

TYPES OF A COMPUTER

Classification of Computers Based on the Principle of Working

1. Analog Computers

Analog computers are used to process continuous data. Analog computers represent variables by physical quantities. Thus any computer which solve problem by translating physical conditions such as flow, temperature, pressure, angular position or voltage into related mechanical or electrical related circuits as an analog for the physical phenomenon being investigated in general it is a computer which uses an analog quantity and produces analog values as output. Thus an analog computer measures continuously. Analog computers are very much speedy. They produce their results very fast. But their results are approximately correct.

2. Digital Computers

Digital computer represents physical quantities with the help of digits or numbers. These numbers are used to perform Arithmetic calculations and also make logical decision to reach a conclusion, depending on, the data they receive from the user.

3. Hybrid Computers

Various specifically designed computers are with both digital and analog characteristics combining the advantages of analog and digital computers when working as a system. Hybrid computers are being used extensively in process control system where it is necessary to have a close representation with the physical world.

Classification of Computers According to Size

1. Super Computers

Large scientific and research laboratories as well as the government organizations have extra ordinary demand for processing data which required tremendous processing speed, memory and other services which may not be provided with any other category to meet their needs. Therefore very large computers used are called Super Computers. These computers are extremely expensive and the speed is measured in billions of instructions per seconds.

2. Main Frame Computers

The most expensive, largest and the most quickest or speedy computer are called mainframe computers. These computers are used in large companies, factories, organizations etc. the mainframe computers are the most expensive computers, they cost more than 20 million rupees. In this computer 150 users are able to work on one C.P.U. The mainframes are able to process 1 to 8 bits at a time. They have several hundreds of megabytes of primary storage and operate at a speed measured in nano second.

3. Mini Computers

Mini computers are smaller than mainframes, both in size and other facilities such as speed, storage capacity and other services. They are versatile that they can be fitted where ever they are needed. Their speeds are rated between one and fifty million instructions per second (MIPS). They have primary storage in hundred to three hundred megabytes range with direct access storage device.

4. Micro Computers

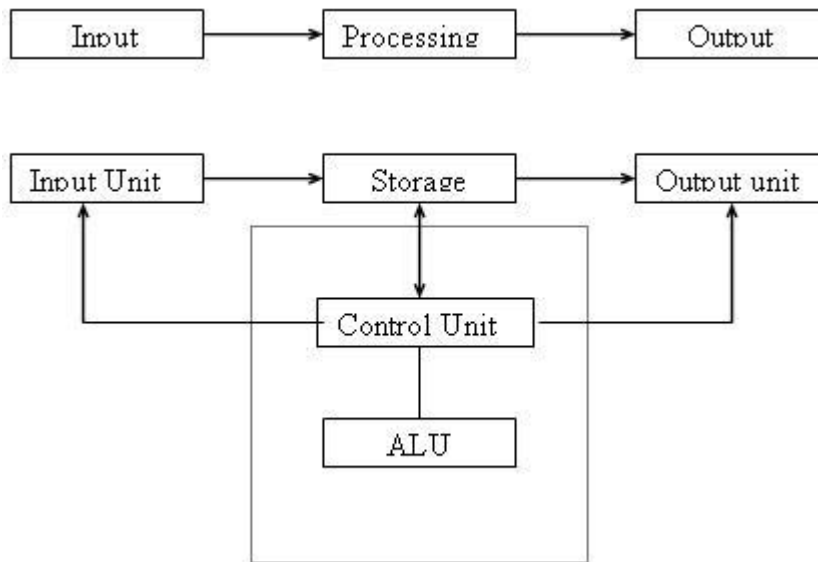
These are the smallest range of computers. They were introduced in the early 70's having less storing space and processing speed. Micro computers of today's are equivalent to the mini computers of yesterday in terms of performing and processing. They are also called "computer of a chip" because its entire circuitry is contained in one tiny chip. The micro computers have a wide range of applications including uses as portable computer that can be plugged into any wall.

5. Laptop Computers

The smallest computer in size has been developed. This type of small computers look like an office brief case and called "LAPTOP" computer. The laptops are also termed as "PORTABLE COMPUTERS." Due to the small size and light weight, they become popular among the computer users. The businessmen found laptop very useful, during traveling and when they are far away from their desktop computers. A typical laptop computer has all the facilities available in microcomputer. The smallest laptops are called "PALMTOP".

ORGANIZATION OF COMPUTER

Block Diagram of Computer:



A computer can process data, pictures, sound and graphics. They can solve highly complicated problems quickly and accurately.

Input Unit:

Computers need to receive data and instruction in order to solve any problem. Therefore we need to input the data and instructions into the computers. The input unit consists of one or more input devices. Keyboard is the one of the most commonly used input device. Other commonly used input devices are the mouse, floppy disk drive, magnetic tape, etc. All the input devices perform the following functions.

- Accept the data and instructions from the outside world.
- Convert it to a form that the computer can understand.
- Supply the converted data to the computer system for further processing.

Storage Unit:

The storage unit of the computer holds data and instructions that are entered through the input unit, before they are processed. It preserves the intermediate and final results before these are sent to the output devices. It also saves the data for the later use. The various storage devices of a computer system are divided into two categories.

1. Primary Storage: Stores and provides very fast. This memory is generally used to hold the program being currently executed in the computer, the data being received from the input unit, the intermediate and final results of the program. The primary memory is temporary in nature. The data is lost, when the computer is switched off. In order to store the data permanently, the data has to be transferred to the secondary memory.

The cost of the primary storage is more compared to the secondary storage. Therefore most computers have limited primary storage capacity.

2. Secondary Storage: Secondary storage is used like an archive. It stores several programs, documents, data bases etc. The programs that you run on the computer are first transferred to the primary memory before it is actually run. Whenever the results are saved, again they get stored in the secondary memory. The secondary memory is slower and cheaper than the primary memory. Some of the commonly used secondary memory devices are Hard disk, CD, etc.

Memory Size:

All digital computers use the binary system, i.e. 0's and 1's. Each character or a number is represented by an 8 bit code. The set of 8 bits is called a byte. A character occupies 1 byte space. A numeric occupies 2 byte space. Byte is the space occupied in the memory.

The size of the primary storage is specified in KB (Kilobytes) or MB (Megabyte). One KB is equal to 1024 bytes and one MB is equal to 1000KB. The size of the primary storage in a typical PC usually starts at 16MB. PCs having 32 MB, 48MB, 128 MB, 256MB memory are quite common.

Output Unit:

The output unit of a computer provides the information and results of a computation to outside world. Printers, Visual Display Unit (VDU) are the commonly used output devices. Other commonly used output devices are floppy disk drive, hard disk drive, and magnetic tape drive.

Arithmetic Logical Unit:

All calculations are performed in the Arithmetic Logic Unit (ALU) of the computer. It also does comparison and takes decision. The ALU can perform basic operations such as addition, subtraction, multiplication, division, etc and does logic operations viz, >, <, =, 'etc. Whenever calculations are required, the control unit transfers the data from storage unit to ALU once the computations are done, the results are transferred to the storage unit by the control unit and then it is send to the output unit for displaying results.

Control Unit:

It controls all other units in the computer. The control unit instructs the input unit, where to store the data after receiving it from the user. It controls the flow of data and instructions from the storage unit to ALU. It also controls the flow of results from the ALU to the storage unit. The control unit is generally referred as the central nervous system of the computer that control and synchronizes it's working.

Central Processing Unit:

The control unit and ALU of the computer are together known as the Central Processing Unit (CPU). The CPU is like brain performs the following functions:

- It performs all calculations.
- It takes all decisions.
- It controls all units of the computer.

A PC may have CPU-IC such as Intel 8088, 80286, 80386, 80486, Celeron, Pentium, Pentium Pro, Pentium II, Pentium III, Pentium IV, Dual Core, and AMD etc

MICROPROCESSOR

A microprocessor incorporates the functions of a computer's central processing unit (CPU) on a single integrated circuit (IC),[1] or at most a few integrated circuits.[2] It is a multi-purpose, programmable device that accepts digital data as input, processes it according to instructions stored in its memory, and provides results as output. It is an example of sequential digital logic, as it has internal memory. Microprocessors operate on numbers and symbols represented in the binary numeral system.

A silicon chip that contains a CPU. In the world of personal computers, the terms microprocessor and CPU are used interchangeably. At the heart of all personal computers and most workstations sits a microprocessor. Microprocessors also control the logic of almost all digital devices, from clock radios to fuel-injection systems for automobiles.

Three basic characteristics differentiate microprocessors:

Instruction set: The set of instructions that the microprocessor can execute.

Instruction types

Some operations available in most instruction sets include:

- **Data handling and Memory operations**
 - **set** a register (a temporary "scratchpad" location in the CPU itself) to a fixed constant value
 - **move** data from a memory location to a register, or vice versa. This is done to obtain the data to perform a computation on it later, or to store the result of a computation
 - **read** and **write** data from hardware devices
- **Arithmetic and Logic**
 - **add**, **subtract**, **multiply**, or **divide** the values of two registers, placing the result in a register, possibly setting one or more condition codes in a status register
 - perform bitwise operations, taking the conjunction and disjunction of corresponding bits in a pair of registers, or the negation of each bit in a register
 - **compare** two values in registers (for example, to see if one is less, or if they are equal)
- **Control flow**
 - branch to another location in the program and execute instructions there
 - conditionally branch to another location if a certain condition holds
 - indirectly branch to another location, while saving the location of the next instruction as a point to return to (a call)

Status Register (SREG)

- SREG: Status Register
- C: Carry Flag
- Z: Zero Flag
- N: Negative Flag
- V: Two's complement overflow indicator
- S: $N \oplus V$, For signed tests
- H: Half Carry Flag
- T: Transfer bit used by BLD and BST instructions
- I: Global Interrupt Enable/Disable Flag

Registers and Operands

- Rd: Destination (and source) register in the Register File
- Rr: Source register in the Register File
- R: Result after instruction is executed
- K: Constant data
- k: Constant address
- b: Bit in the Register File or I/O Register (3-bit)
- s: Bit in the Status Register (3-bit)
- X,Y,Z: Indirect Address Register
- (X=R27:R26, Y=R29:R28 and Z=R31:R30)
- A: I/O location address
- q: Displacement for direct addressing (6-bit)

Bandwidth: The number of bits processed in a single instruction.

Clock speed: Given in megahertz (MHz), the clock speed determines how many instructions per second the processor can execute.

In both cases, the higher the value, the more powerful the CPU. For example, a 32-bit microprocessor that runs at 50MHz is more powerful than a 16-bit microprocessor that runs at 25MHz.

In addition to bandwidth and clock speed, microprocessors are classified as being either RISC (reduced instruction set computer) or CISC (complex instruction set computer).

THE EVOLUTION OF MICROPROCESSOR

The evolution of microprocessors has been broadly divided into 4 generations. This is described below.

First generation microprocessors-

The first generation of microprocessors was introduced in 1971-1973. They used monolithic IC and P-MOS technology. This technology has slow speed, not supported with TTL. Due to the lack of pins, signals have to be multiplexed.

Example-4004 & 4040

Second generation microprocessors-

These microprocessors were introduced in 1973-1978. They were designed using N-MOS technology. Faster speed and higher density packaging than P-MOS technology. it had more powerful instruction set, ability to handle large memory space and had better interrupt handling capability.

Example-8080 & 8085

Third generation microprocessors-

These were introduced in between 1978-1980. They were basically 16 bit processor and were made by using H-MOS technology. It had flexible input-output port addresses.

Example- 8 bit-8086

16 bit- 80286, 80386 & 80486.

Fourth generation microprocessor-

These were introduced after 1980. It had 2kb main memory, 16 mb physical memory, 1 tb virtual memory for enhancing speed.

Example- Celeron and further invented processors.

Reduced instruction set computing

Reduced instruction set computing, or **RISC** is a CPU design strategy based on the insight that simplified (as opposed to complex) instructions can provide higher performance if this simplicity enables much faster execution of each instruction. A computer based on this strategy is a *reduced instruction set computer* also called *RISC*.

A number of systems, going back to the 1970s (and even 1960s) have been credited as the first RISC architecture, partly based on their use of load/store approach. The term RISC was coined by David Patterson of the Berkeley RISC project, although somewhat similar concepts had appeared before.

RISC-Means Reduced Instruction Set Computer. a Risc system has reduced number of instructions and more importantly it is load store architecture where pipelining can be implemented easily. E.g. ATMEL AVR

- Instruction Set: large set of instruction with variable size (16 to 64)
- Addressing Modes: 12-24
- General Purpose registers: 8-24
- Clock rate: 33-50MHz in 1992

Complex instruction set computing

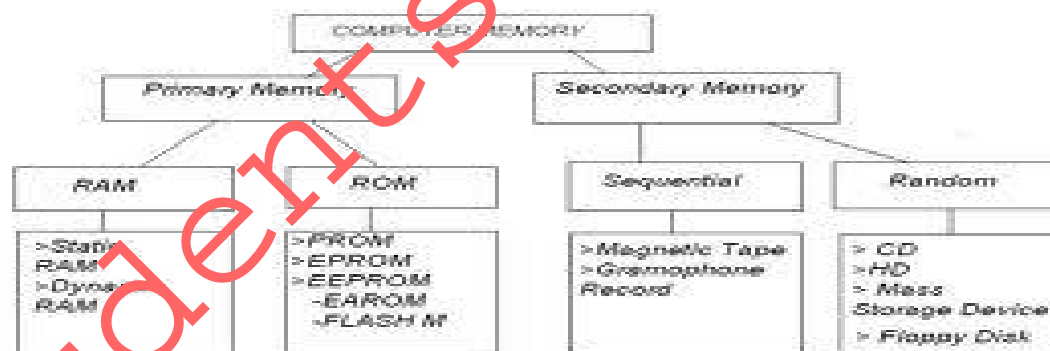
A **complex instruction set computer (CISC)**, is a computer where single instructions can execute several low-level operations (such as a load from memory, an arithmetic operation, and a memory store) and/or are capable of multi-step operations or addressing modes within single instructions. The term was retroactively coined in contrast to reduced instruction set computer (RISC).

Examples of CISC instruction set architectures are System/360 through z/Architecture, PDP-11, VAX, Motorola 68k, and x86.

CISC-Means Complex instruction set architecture. A CISC system 8085 has complex instructions such as direct addition between data in two memory locations. E.g.

- Instruction Set: Small set of instruction with fixed size (32-bit)
- Addressing Modes: 3-5
- General Purpose registers: 32-192
- Clock rate: 50-150MHz in 1993

MEMORY UNIT



The memory unit is the unit where all the input data and results stored. The CPU memory is also called as memory register. The memory of a computer is also available in the form of Random Access Memory (RAM). RAM is a semiconductor chip. RAM is considered as a volatile memory, it means as long power is supporting information stored in it remain. Once the power is lost, the information stored in the RAM also gets erased. Microcomputers contains read Only Memory (ROM). ROM contains instructions for the microcomputers.

Microcomputers use ROM, programmable read only memory (PROM), and erasable programmable read-only memory (EPROM) to store selected application programs. The contents of ROM are determined when the chips are manufactured. The ROM memory is considered as non volatile, means the information is not get erased even when power is failed. The most important ROM chip(s) we should know about is the Basic Input/output system or BIOS. The BIOS is a collection of small computer programs built into a ROM chip.

On personal computer there are three types of memory. They are

Conventional memory: The memory into which we load our software and work files. Conventional memory also known as base or low memory is any memory below 1M (1024) although only 640k of it is directly available for our work.

Extended memory (XMS): Memory above 1M. This type of memory is usually not directly available to our software.

Expanded memory (EMS): To expand the memory by reserving a special peephole of 64kb of memory to be used when the computer requests certain data not immediately available from RAM. Usually a software utility called an Expanded Memory Manager (EMM) manages this expanded memory.

Computer Memory is classified into:

- **Main memory**
- **Secondary memory**
- **Cache memory**

MAIN MEMORY

Refers to physical memory that is internal to the computer. The word main is used to distinguish it from external mass storage devices such as disk drives. Another term for main memory is RAM.

The computer can manipulate only data that is in main memory. Therefore, every program you execute and every file you access must be copied from a storage device into main memory. The amount of main memory of a computer is crucial because it determines how many programs can be executed at one time and how much data can be readily available to a program.

Because computers often have too little main memory to hold all the data they need, computer engineers invented a technique called swapping, in which portions of data are copied into main memory as they are needed. Swapping occurs when there is no room in memory for needed data. When one portion of data is copied into memory, an equal-sized portion is copied (swapped) out to make room.

RAM

Random access memory (RAM) is a form of computer data storage. A random access device allows stored data to be accessed in any order in very nearly the same amount of time for any storage location or size of memory device. A device such as a magnetic tape requires increasing

time to access data stored on parts of the tape that are far from the ends. Memory devices (such as floppy discs, CDs and DVDs) can access the storage data only in a predetermined order, because of mechanical design limitations; the time to access a given part of the device varies significantly due to its physical location.

ROM

Read-only memory (ROM) is a class of storage medium used in computers and other electronic devices. Data stored in ROM cannot be modified, or can be modified only slowly or with difficulty, so it is mainly used to distribute firmware (software that is very closely tied to specific hardware and unlikely to need frequent updates).

In its strictest sense, ROM refers only to mask ROM (the oldest type of solid state ROM) which is fabricated with the desired data permanently stored in it, and thus can never be modified. Despite the simplicity, speed and economies of scale of mask ROM, field-programmability often make reprogrammable memories more flexible and inexpensive. As of 2007, actual ROM circuitry is therefore mainly used for applications such as microcode, and similar structures, on various kinds of processors.

PROM

A programmable read-only memory (PROM) or field programmable read-only memory (FPGA) or one-time programmable non-volatile memory (OTP NVM) is a form of digital memory where the setting of each bit is locked by a fuse or antifuse. Such PROMs are used to store programs permanently. The key difference from a strict ROM is that the programming is applied after the device is constructed.

PROMs are manufactured blank and, depending on the technology, can be programmed at wafer, final test, or in system. The availability of this technology allows companies to keep a supply of blank PROMs in stock, and program them at the last minute to avoid large volume commitment. These types of memories are frequently seen in video game consoles, mobile phones, radio-frequency identification (RFID) tags, implantable medical devices, high-definition multimedia interfaces (HDMI) and in many other consumer and automotive electronics products.

EPROM

An EPROM (rarely EPROM), or erasable programmable read only memory, is a type of memory chip that retains its data when its power supply is switched off. In other words, it is non-volatile. It is an array of floating-gate transistors individually programmed by an electronic device that supplies higher voltages than those normally used in digital circuits. Once programmed, an EPROM can be erased by exposing it to strong ultraviolet light source (such as from a mercury-vapor light). EPROMs are easily recognizable by the transparent fused quartz window in the top of the package, through which the silicon chip is visible, and which permits exposure to UV light during erasing.

EEPROM

EEPROM stands for Electrically Erasable Programmable Read-Only Memory and is a type of non-volatile memory used in computers and other electronic devices to store small amounts of data that must be saved when power is removed, e.g., calibration tables or device configuration.

When larger amounts of static data are to be stored (such as in USB flash drives) a specific type of EEPROM such as flash memory is more economical than traditional EEPROM devices. EEPROMs are realized as arrays of floating-gate transistors.

EEPROM is user-modifiable read-only memory (ROM) that can be erased and reprogrammed (written to) repeatedly through the application of higher than normal electrical voltage generated externally or internally in the case of modern EEPROMs. EPROM usually must be removed from the device for erasing and programming, whereas EEPROMs can be programmed and erased in circuit. Originally, EEPROMs were limited to single byte operations which made them slower, but modern EEPROMs allow multi-byte page operations.

SECONDARY MEMORY

Secondary memory is where programs and data are kept on a long-term basis. Common secondary storage devices are the hard disk and floppy disks.

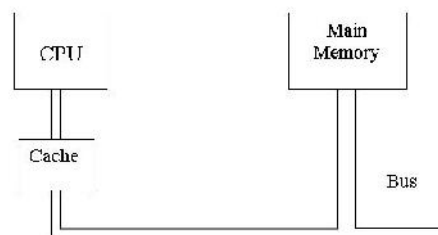
- The hard disk has enormous storage capacity compared to main memory.
- The hard disk is usually contained in the systems unit of a computer.
- The hard disk is used for long-term storage of programs and data.
- Data and programs on the hard disk are organized into files-named sections of the disk.

A hard disk might have a storage capacity of 40 gigabytes. This is about 300 times the amount of storage in main memory (assuming 128 megabytes of main memory.) However, a hard disk is very slow compared to main memory. The reason for having two types of storage is this contrast:

Primary memory	Secondary memory
<ol style="list-style-type: none"> 1. Fast 2. Expensive 3. Low capacity 4. Connects directly to the processor 	<ol style="list-style-type: none"> 1. Slow 2. Cheap 3. Large capacity 4. Not connected directly to the processor

Floppy disks are mostly used for transferring software between computer systems and for casual backup of software. They have low capacity, and are very, very slow compared to other storage devices.

CACHE MEMORY



A CPU cache is a cache used by the central processing unit of a computer to reduce the average time to access memory. The cache is a smaller, faster memory which stores copies of the data from the most frequently used main memory locations. As long as most memory accesses are cached memory locations, the average latency of memory accesses will be closer to the cache latency than to the latency of main memory.

INPUT AND OUTPUT DEVICES

I. Introduction

The computer will be of no use unless it is able to communicate with the outside World. Input/output devices are required for users to communicate with the computer. In simple terms, input devices bring information INTO the computer and output Devices bring information OUT of a computer system. These input/output devices are also known as peripherals since they surround the CPU and memory of a computer system.

II. Input Devices

(a) Keyboard

It is a text base input device that allows the user to input alphabets, numbers and Other **characters**. It consists of a set of keys mounted on a board.



Figure 1: The Keyboard and Mouse

Alphanumeric Keypad

It consists of keys for English alphabets, 0 to 9 numbers, and special characters like + - / * () etc.

Function Keys

There are twelve function keys labeled F1, F2, F3... F12. The functions assigned to these keys differ from one software package to another. These keys are also user programmable keys.

Special-function Keys

These keys have special functions assigned to them and can be used only for those specific purposes. Functions of some of the important keys are defined below.

Enter

It is similar to the 'return' key of the typewriter and is used to execute a command or program.

Spacebar

It is used to enter a space at the current cursor location.

Backspace

This key is used to move the cursor one position to the left and also delete the character in that position.

Delete

It is used to delete the character at the cursor position.

Insert

Insert key is used to toggle between insert and overwrite mode during data entry.

Shift

This key is used to type capital letters when pressed along with an alphabet key. Also used to type the special characters located on the upper-side of a key that has two characters defined on the same key.

Caps Lock

Cap Lock is used to toggle between the capital lock features. When 'on', it locks the alphanumeric keypad for capital letters input only.

Tab

Tab is used to move the cursor to the next tab position defined in the document. Also, it is used to insert indentation into a document.

Ctrl

Control key is used in conjunction with other keys to provide additional functionality on the keyboard.

Alt

Also like the control key, Alt key is always used in combination with other keys to perform specific tasks.

Esc

This key is usually used to negate a command. Also used to cancel or abort executing programs.

Numeric Keypad

Numeric keypad is located on the right side of the keyboard and consists of keys having numbers (0 to 9) and mathematical operators (+ – * /) defined on them. This keypad is provided to support quick entry for numeric data.

Cursor Movement Keys

These are arrow keys and are used to move the cursor in the direction indicated by the arrow (up, down, left, right).

(b) Mouse

The mouse is a small device used to point to a particular place on the screen and select in order to perform one or more actions. It can be used to select menu commands, size windows, start programs etc. The most conventional kind of mouse has two buttons on top: the left one being used most frequently.

Mouse Actions

Left Click: Used to select an item.

Double Click: Used to start a program or open a file.

Right Click: Usually used to display a set of commands.

Drag and Drop: It allows you to select and move an item from one location to another. To achieve this place the cursor over an item on the screen, click the left Mouse button and while holding the button down move the cursor to where you want to place the item, and then release it.

(c) Joystick

The joystick is a vertical stick which moves the graphic cursor in a direction the stick is moved. It typically has a button on top that is used to select the option pointed by the cursor. Joystick is used as an input device primarily used with video games, Training simulators and controlling robots



Figure 3: The Joystick

(d) Scanner

Scanner is an input device used for direct data entry from the source document into the computer system. It converts the document image into digital form so that it can be fed into the computer. Capturing information like this reduces the possibility of errors typically experienced during large data entry.



Figure 4: The Scanner

Hand-held scanners are commonly seen in big stores to scan codes and price information for each of the items. They are also termed the bar code readers.

(e) Bar codes

A bar code is a set of lines of different thicknesses that represent a number. Bar Code Readers are used to input data from bar codes. Most products in shops have bar codes on them. Bar code readers work by shining a beam of light on the lines that make up the bar code and detecting the amount of light that is reflected back.



Figure 5: The Bar Code Reader

(f) Light Pen

It is a pen shaped device used to select objects on a display screen. It is quite like the mouse but uses a light pen to move the pointer and select any object on the screen by pointing to the object. Users of Computer Aided Design (CAD) applications commonly use the light pens to directly draw on screen.

(g) Touch Screen

It allows the user to operate/make selections by simply touching the display screen. Common examples of touch screen include information kiosks, and bank ATMs.

(h) Digital camera

A digital camera can store many more pictures than an ordinary camera. Pictures taken using a digital camera are stored inside its memory and can be transferred to a computer by connecting the camera to it. A digital camera takes pictures by converting the light passing through the lens at the front into a digital image.



Figure 6: The Digital camera

(i) The Speech Input Device

The “Microphones - Speech Recognition” is a speech Input device. To operate it we require using a microphone to talk to the computer. Also we need to add a sound card to the computer. The Sound card digitizes audio input into 0/1s .A speech recognition program can process the input and convert it into machine-recognized commands or input.



Figure 7: The Microphone

III. Output Devices

(a) Monitor

Monitor is an output device that resembles the television screen and uses a Cathode Ray Tube (CRT) to display information. The monitor is associated with a keyboard for manual input of characters and displays the information as it is keyed in. It also displays the program or application output. Like the television, monitors are also available in different sizes.

(b) Liquid Crystal Display (LCD)

LCD was introduced in the 1970s and is now applied to display terminals also. Its advantages like low energy consumption, smaller and lighter have paved its way for usage in portable computers (laptops).



Figure 8: The LCD

(c) Printer

Printers are used to produce paper (commonly known as hardcopy) output. Based on the technology used, they can be classified as Impact or Non-impact printers. Impact printers use the type writing printing mechanism wherein a hammer strikes the paper through a ribbon in order to produce output. Dot-matrix and Character printers fall under this category. Non-impact printers do not touch the paper while printing. They use chemical, heat or electrical signals to etch the symbols on paper. Inkjet, Deskjet, Laser, Thermal printers fall under this category of printers. When we talk about printers we refer to two basic qualities associated with printers: resolution, and speed. Print resolution is measured in terms of number of dots per inch (dpi). Print speed is

measured in terms of number of characters printed in a unit of time and is represented as characters-per-second (cps), lines-per-minute (lpm), or pages-per-minute (ppm).



Figure 9: The Printer

Types of Printer

Many of the printing methods described in this section are now obsolete but are included for those interested in the history of computing technology!

Line Printer

Line printers have a spinning horizontal drum that stretches the full width of the paper which is separated from it by an inked ribbon. The drum is made up of 132 thin cylinders each having a complete set of characters. Behind the paper is a row of 132 hammers that strike the paper at the right moment to select the required character from the corresponding cylinder. In this way it is able to print a complete line at a time. Line printers are used for high volume low quality output and are very noisy. They are obsolete now.

Dot Matrix

Dot matrix printers have a horizontally moving head with a vertical line of pins mounted inside. An inked ribbon is located between the head and the paper and as the head moves the pins strike the ribbon to form each character as a series of dots. The best quality printers have heads with 24 pins and low quality ones have 9 pins (although by making two passes and shifting the head half a pin pitch between them they can effectively act as a 18 pin head). As these printers can produce small dots anywhere on the paper most support graphics and have software fonts. Dot matrix printers are quite noisy but can be cheap. They are mostly for low to medium quality, low volume personal use.

Daisy Wheel

On a daisy-wheel printer the complete set of characters is held on a removable wheel which consists of a central collar radiating out from which are a set of spokes, each ending in a single character. The wheel spins round to align the required character with a single hammer. The hammer and wheel assembly move across the paper striking it through an inked ribbon. These printers can produce high quality output but are limited to the range of characters on the wheel. Changing wheels is simple, this provides alternative fonts, but is no substitute if a wide range of fonts is required. They are quiet noisy and are used for low volume office work.

Ink Jet

Ink jet printers have a movable head that can spray fine drops of ink directly on to the paper. Some have multiple heads carrying coloured inks with the best ones providing a wide range of colours. As the paper and the head never come into contact they are very quiet. However they are also rather slow. They are generally cheaper than laser printers and are suitable for all types of high quality low volume work. We have a few in the department including one colour Postscript one CPS2.

Laser

In a laser printer, paper is given an electro-static charge by passing it over a charged drum and then a laser scans it discharging all clear areas. Next the paper is passed over a tray of powdered ink (toner) which is attracted to the charged areas. Finally the ink is bonded to the paper by heat or pressure. Laser printers are quiet and are used for high quality low or high volume work. We have a number in the department including PS9 (a Postscript printer near 615) and PS4 (a double sided Postscript printer outside room 663).

Camera Copy

In this case laser light writes directly onto film inside a camera to make very high quality full colour output. Again the process is expensive and the volume low. OUCS have a camera copy service that takes Postscript format files.

(d) Plotter

Plotters are used to print graphical output on paper. It interprets computer commands and makes line drawings on paper using multicoloured automated pens. It is capable of producing graphs, drawings, charts, maps etc. Computer Aided Engineering (CAE) applications like CAD (Computer Aided Design) and CAM (Computer Aided Manufacturing) are typical usage areas for plotters.

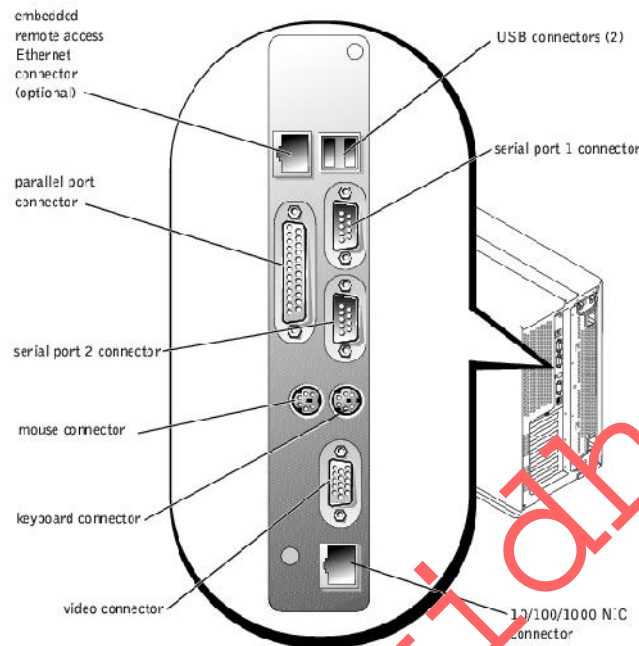


Figure 10: The Plotter

(e) Audio Output: Sound Cards and Speakers:

The Audio output is the ability of the computer to output sound. Two components are needed: Sound card – Plays contents of digitized recordings, Speakers – Attached to sound card.

INPUT/OUTPUT PORT AND CONNECTOR



Serial Ports and Parallel Port

The integrated serial ports use 9-pin D-subminiature connectors on the back panel. These ports support devices such as external modems, printers, plotters, and mice that require serial data transmission (the transmission of data one bit at a time over one line).

Most software uses the term COM (for communications) plus a number to designate a serial port (for example, COM1 or COM2). The default designations of your system's integrated serial ports are COM1 and COM2. The integrated parallel port uses a 25-pin D-subminiature connector on the system's back panel. This I/O port sends data in parallel format (where eight data bits, or one byte, are sent simultaneously over eight separate lines in a single cable). The parallel port is used primarily for printers.

Most software uses the term LPT (for line printer) plus a number to designate a parallel port (for example, LPT1). The default designation of the system's integrated parallel port is LPT1.

Port designations are used, for example, in software installation procedures that include a step in which you identify the port to which a printer is attached, thus telling the software where to send its output. (An incorrect designation prevents the printer from printing or causes scrambled print.)

Serial Port Connector

If you reconfigure your hardware, you may need pin number and signal information for the serial port connector. Table B-1 defines the pin assignments and interface signals for the serial port connector.

Table B-1. Serial Port Pin Assignments

Pin	Signal	I/O	Definition
1	DCD	I	Data carrier detect
2	SIN	I	Serial input
3	SOUT	O	Serial output
4	DTR	O	Data terminal ready
5	GND	N/A	Signal ground
6	DSR	I	Data set ready
7	RTS	O	Request to send
8	CTS	I	Clear to send
9	RI	I	Ring indicator
Shell	N/A	N/A	Chassis ground

Parallel Port Connector

If you reconfigure your hardware, you may need pin number and signal information for the parallel port connector. Table B-2 defines the pin assignments and interface signals for the parallel port connector.

Table B-2. Parallel Port Pin Assignments

Pin	Signal	I/O	Definition
-----	--------	-----	------------

1	STB#	I/O	Strobe
2	PD0	I/O	Printer data bit 0
3	PD1	I/O	Printer data bit 1
4	PD2	I/O	Printer data bit 2
5	PD3	I/O	Printer data bit 3
6	PD4	I/O	Printer data bit 4
7	PD5	I/O	Printer data bit 5
8	PD6	I/O	Printer data bit 6
9	PD7	I/O	Printer data bit 7
10	ACK#	I	Acknowledge
11	BUSY	I	Busy
12	PE	I	Paper end
13	SLCT	I	Select
14	AFD#	O	Automatic feed
15	ERR#	I	Error
16	INIT#	O	Initialize printer
17	SLIN#	O	Select in
18-25	GND	N/A	Signal ground

Adding an Expansion Card Containing Serial or Parallel Ports

The system has an auto configuration capability for the serial ports. This feature lets you add an expansion card containing a serial port that has the same designation as one of the integrated ports, without having to reconfigure the card. When the system detects the duplicate serial port on the expansion card, it remaps (reassigns) the integrated port to the next available port designation.

Both the new and the remapped COM ports share the same IRQ setting, as follows:

COM1, COM3: IRQ4 (shared setting)

COM2, COM4: IRQ3 (shared setting)

These COM ports have the following I/O address settings:

COM1:3F8h

COM2:2F8h

COM3:3E8h

COM4:2E8h

For example, if you add an internal modem card with a port configured as COM1, the system then sees logical COM1 as the address on the modem card. It automatically remaps the integrated serial port that was designated as COM1 to COM3, which shares the COM1 IRQ setting. (Note that when you have two COM ports sharing an IRQ setting, you can use either port as necessary but you may not be able to use them both at the same time.) If you install one or more expansion cards with serial ports designated as COM1 and COM3, the corresponding integrated serial port is disabled.

Before adding a card that remaps the COM ports, check the documentation that accompanied your software to make sure that the software can be mapped to the new COM port designation.

To avoid auto configuration, you may be able to reset jumpers on the expansion card so that the card's port designation changes to the next available COM number, leaving the designation for the integrated port as is. Alternatively, you can disable the integrated ports through the System Setup program. The documentation for your expansion card should provide the card's default I/O address and allowable IRQ settings. It should also provide instructions for readdressing the port and changing the IRQ setting, if necessary.

For general information on how your operating system handles serial and parallel ports, and for more detailed command procedures, see your operating system documentation.

Keyboard and Mouse Connectors

The system uses a PS/2-style keyboard and supports a PS/2-compatible mouse. Cables from both devices attach to 6-pin, miniature DIN connectors on the back panel of your system.

Keyboard Connector

The following information is pin information for the keyboard connector. Figure B-4 illustrates the pin numbers for the keyboard connector. Table B-3 defines the pin assignments and interface signals for the keyboard connector.

Figure B-4. Pin Numbers for the Keyboard Connector



Table B-3. Keyboard Connector Pin Assignments

Pin	Signal	I/O	Definition
1	KBDATA	I/O	Keyboard data
2	NC	N/A	No connection
3	GND	N/A	Signal ground
4	FVcc	N/A	Fused supply voltage
5	KBCLK	I/O	Keyboard clock
6	NC	N/A	No connection
Shell	N/A	N/A	Chassis ground

Mouse Connector

The following is pin information for the mouse connector. Figure B-5 illustrates the pin numbers for the mouse connector. Table B-4 defines the pin assignments and interface signals for the mouse connector.

Table B-4. Mouse Connector Pin Assignments (Back Panel)

Pin	Signal	I/O	Definition
-----	--------	-----	------------

1	MSDATA	I/O	Mouse data
2	NC	N/A	No connection
3	GND	N/A	Signal ground
4	FVcc	N/A	Fused supply voltage
5	MSCLK	I/O	Mouse clock
6	NC	N/A	No connection
Shell	N/A	N/A	Chassis ground

Video Connector

The system uses a 15-pin high-density D-subminiature connector on the front and back panels for attaching a VGA-compatible monitor to your system. The video circuitry on the system board synchronizes the signals that drive the red, green, and blue electron guns in the monitor.

Figure B-6. Pin Numbers for the Video Connector

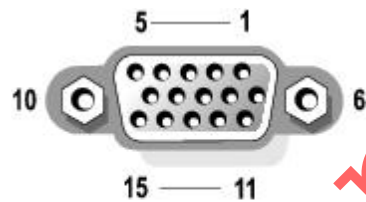


Table B-5. Video Connector Pin Assignments

Pin	Signal	I/O	Definition
1	RED	O	Red video
2	GREEN	O	Green video
3	BLUE	O	Blue video

4	NC	N/A	No connection
5–8, 10	GND	N/A	Signal ground
9	VCC	N/A	Vcc
11	NC	N/A	No connection
12	DDC data out	O	Monitor detect data
13	HSYNC	O	Horizontal synchronization
14	VSNC	O	Vertical synchronization
15	DDC clock out	O	Monitor detect clock
Shell	N/A	N/A	Chassis ground

USB Connectors

Your system contains two USB connectors on the rear panel for attaching USB-compliant devices. USB devices are typically peripherals such as mice, keyboards, and system speakers.

NOTICE: Do not attach a USB device or a combination of USB devices that draw a maximum current over 500 mA per channel on +5 V. Attaching devices that exceed this threshold may cause the USB ports to shut down. See the documentation that accompanied the USB devices for their maximum current ratings.

If you reconfigure your hardware, you may need pin number and signal information for the USB connectors. Figure B-7 illustrates the USB connector and Table B-6 defines the pin assignments and interface signals for the USB connector.

Figure B-7. Pin Numbers for the USB Connector



Table B-6. USB Connector Pin Assignments

Pin	Signal	I/O	Definition
1	Vcc	N/A	Supply voltage
2	DATA-	I/O	Data
3	DATA+	I/O	Data
4	GND	N/A	Signal ground

Integrated NIC Connector

Your system has one integrated 10/100/1000 Mbps NIC. The 10/100/1000 Mbps NIC connectors provide faster communication between servers and workstations and efficient utilization of host resources, freeing more of the system resources for other applications. The NIC supports 10 Base-T, 100 Base-TX, and 1000 Base-T Ethernet standards.

The NIC includes a Wake On LAN feature that enables the system to be started by a special LAN signal from a systems management console. Wake On LAN provides remote system setup, software downloading and installation, file updates, and asset tracking after hours and on weekends when LAN traffic is typically at a minimum.

Network Cable Requirements

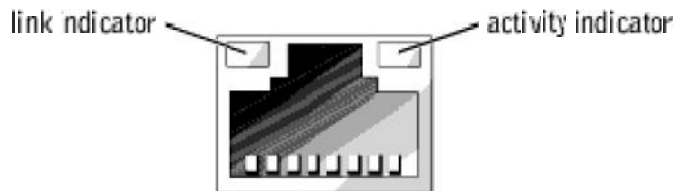
Your system's RJ45 NIC connector is designed for attaching a UTP Ethernet cable equipped with standard RJ45-compatible plugs. Press one end of the UTP cable into the NIC connector until the plug snaps securely into place. Connect the other end of the cable to an RJ45 jack wall plate or to an RJ45 port on a UTP concentrator or hub, depending on your network configuration. Observe the following cabling restrictions for 10 Base-T, 100 Base-TX, and 1000 Base-T networks.

- Use Category 5 or greater wiring and connectors.
- The maximum cable run length (from a system to a hub) is 328 ft (100 m).
- You can find guidelines for operation of a network can be found in "Systems Considerations of Multi-Segment Networks" in the IEEE 802.3 standard.

Embedded Remote Access Ethernet Connector (Optional)

Your system's optional embedded remote access Ethernet connector is designed to provide remote access capabilities for your system. It is designed specifically to work with systems management software.

Figure B-9. Embedded Remote Access Ethernet Connector



OPERATING SYSTEM

An operating system is a layer of software which takes care of technical aspects of a computer's operation. It shields the user of the machine from the low-level details of the machine's operation and provides frequently needed facilities. There is no universal definition of what an operating system consists of. You can think of it as being the software which is already installed on a machine, before you add anything of your own. Normally the operating system has a number of key elements:

- (i) A *technical layer of software* for driving the hardware of the computer, like disk drives, the keyboard and the screen;
- (ii) A *file system* which provides a way of organizing files logically, and
- (iii) A simple *command language* which enables users to run their own programs and to manipulate their files in a simple way. Some operating systems also provide text editors, compilers, debuggers and a variety of other tools. Since the operating system (OS) is in charge of a computer, all requests to use its resources and devices need to go through the OS.
- (iv) An OS therefore provides *Legal entry points* into its code for performing basic operations like writing to devices.

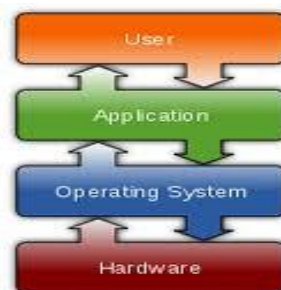


Fig. Operating System

Operating systems may be classified by both how many tasks they can perform 'simultaneously' and by how many users can be using the system 'simultaneously'. That is: *single-user* or *multi-user* and *single-task* or *multi-tasking*. A multi-user system must clearly be multi-tasking.

Functions of Operating System

An operating system (OS) is a set of computer program that manages the hardware and software resources of a computer. At the foundation of all system software, the OS performs basic tasks such as controlling and allocating memory, prioritizing system requests, controlling input and output devices, facilitating networking, and managing files. It also may provide a graphical user interface for higher level functions. Various services performed by operating systems are discussed below.

1. **Process management:** It deals with running multiple processes. Most operating system allows a process to be assigned a priority which affects its allocation of CPU time. Interactive operating systems also employ some level of feedback in which the task with which the user is working receives higher priority. In many systems there is a background process which runs when no other process is waiting for the CPU.
2. **Memory management:** The memory manager in an OS coordinates the memories by tracking which one is available, which is to be allocated or deal located and how to swap between the main memory and secondary memories. The operating system tracks all memory used by each process so that when a process terminates, all memory used by that process will be available for other processes.
3. **Disk and file systems:** Operating systems have a variety of native file systems that controls the creation, deletion, and access of files of data and programs.
4. **Networking:** Most current operating systems are capable of using the TCP/IP networking protocols. This means that one system can appear on a network of the other and share resources such as files, printers, and scanners. Many operating systems also support one or more vendor-specific legacy networking protocols as well.
5. **Security:** most operating systems include some level of security.
6. **Device drivers:** A device driver is a specific type of computer software developed to allow interaction with hardware devices. Typically this constitutes an interface for communicating with the device, through the specific computer bus or communications subsystem that the hardware is connected to, providing commands to and/or receiving data from the device, and on the other end, the requisite interfaces to the operating system and software applications.

Types of Operating System

Operating system is a platform between hardware and user which is responsible for the management and coordination of activities and the sharing of the resources of a computer. It hosts the several applications that run on a computer and handles the operations of computer hardware. There are some different operating systems. These are as follows:

1. Real-time Operating System: It is a multitasking operating system that aims at executing real-time applications.

2. Multi-user and Single-user Operating Systems: The operating systems of this type allow a multiple users to access a computer system concurrently.

3. Multi-tasking and Single-tasking Operating Systems: When a single program is allowed to run at a time, the system is grouped under a single-tasking system, while in case the operating system allows the execution of multiple tasks at one time, it is classified as a multi-tasking operating system.

4. Distributed Operating System: An operating system that manages a group of independent computers and makes them appear to be a single computer is known as a distributed operating system.

5. Embedded System: The operating systems designed for being used in embedded computer systems are known as embedded operating systems.

Types of Operating System

- DOS (Disk Operating System)
- UNIX
- LINUX
- Window XP

DOS (Disk Operating System)

DOS (Disk Operating System) was the first widely-installed operating system for personal computers. (Earlier, the same name had been used for an IBM operating system for a line of business computers.)

The first personal computer version of DOS, called **PC-DOS**, was developed for IBM by Bill Gates and his new Microsoft Corporation. He retained the rights to market a Microsoft version, called **MS-DOS**. PC-DOS and MS-DOS are almost identical and most users have referred to either of them as just "DOS." DOS was (and still is) a non-graphical line-oriented command- or menu-driven operating system, with a relatively simple interface but not overly "friendly" user interface. Its prompt to enter a command looks like this:

```
C>
```

The first Microsoft Windows operating system was really an application that ran on top of the MS-DOS operating system. Today, Windows operating systems continue to support DOS (or a DOS-like user interface) for special purposes by emulating the operating system.

In the 1970s before the personal computer was invented, IBM had a different and unrelated DOS (Disk Operating System) that ran on smaller business computers. It was replaced by IBM's VSE operating system.

Advantages:

1. DOS is very lightweight and it allows direct access to most hardware.
2. It does not have the overhead of a multitasking operating system.

Disadvantages:

1. It is 16-bit and limited to 640k of RAM (this can be overcome with a DOS extender).
2. It runs in real mode, so a buggy or malicious program can cause corruption.

UNIX Operating System

UNIX (officially trademarked as UNIX, sometimes also written as Unix) is a multitasking, multi-user computer operating system originally developed in 1969 by a group of AT&T employees at Bell Labs, including Ken Thompson, Dennis Ritchie, Brian Kernighan, Douglas McIlroy, Michael Lesk and Joe Ossanna. The UNIX operating system was first developed in assembly language, but by 1973 had been almost entirely recoded in C, greatly facilitating its further development and porting to other hardware. Today's Unix system evolution is split into various branches, developed over time by AT&T as well as various commercial vendors, universities (such as University of California, Berkeley's BSD), and non-profit organizations.

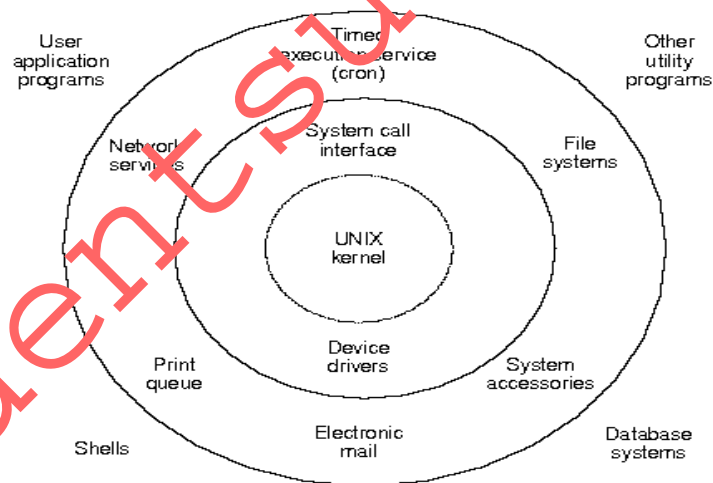


Fig. Unix Operating System

The Open Group, an industry standards consortium, owns the UNIX trademark. Only systems fully compliant with and certified according to the Single UNIX Specification are qualified to use the trademark; others might be called UNIX system-like or Unix-like, although the Open Group disapproves of this term. However, the term UNIX is often used informally to denote any operating system that closely resembles the trademarked system.

During the late 1970s and early 1980s, the influence of Unix in academic circles led to large-scale adoption of Unix (particularly of the BSD variant, originating from the University of California, Berkeley) by commercial startups, the most notable of which are Solaris, HP-UX, Sequent, and AIX, as well as Darwin, which forms the core set of components upon which Apple's OS X, Apple TV, and iOS are based. Today, in addition to certified UNIX systems such as those already mentioned, Unix-like operating systems such as MINIX, Linux, Android, and BSD descendants (FreeBSD, NetBSD, OpenBSD, and DragonFly BSD) are commonly encountered. The term traditional UNIX may be used to describe an operating system that has the characteristics of either Version 7 Unix or UNIX System V.

Advantages of UNIX

1. UNIX is more flexible and can be installed on many different types of machines, including main-frame computers, supercomputers and micro-computers.
2. UNIX is more stable and does not go down as often as Windows does, therefore requires less administration and maintenance.
3. UNIX has greater built-in security and permissions features than Windows.
4. UNIX possesses much greater processing power than Windows.
5. UNIX is the leader in serving the Web. About 90% of the Internet relies on UNIX operating systems running Apache, the world's most widely used Web server.
6. Software upgrades from Microsoft often require the user to purchase new or more hardware or prerequisite software. That is not the case with UNIX.
7. The mostly free or inexpensive open-source operating systems, such as Linux and BSD, with their flexibility and control, are very attractive to (aspiring) computer wizards. Many of the smartest programmers are developing state-of-the-art software free of charge for the fast growing "open-source movement".
8. UNIX also inspires novel approaches to software design, such as solving problems by interconnecting simpler tools instead of creating large monolithic application programs.

Disadvantages of UNIX

1. The traditional command line shell interface is user hostile -- designed for the programmer, not the casual user.
2. Commands often have cryptic names and give very little response to tell the user what they are doing. Much use of special keyboard characters - little typos have unexpected results.
3. To use Unix well, you need to understand some of the main design features. Its power comes from knowing how to make commands and programs interact with each other, not just from treating each as a fixed black box.
4. Richness of utilities (over 400 standard ones) often overwhelms novices. Documentation is short on examples and tutorials to help you figure out how to *use* the many tools provided to accomplish various kinds of tasks.

Linux Operating System

Linux was originally developed as a free operating system for Intel x86-based personal computers. It has since been ported to more computer hardware platforms than any other operating system. It is a leading operating system on servers and other big iron systems such as mainframe computers and supercomputers: more than 90% of today's 500 fastest supercomputers run some variant of Linux, including the 10 fastest. Linux also runs on embedded systems (devices where the operating system is typically built into the firmware and highly tailored to the system) such as mobile phones, tablet computers, network routers, televisions and video game consoles; the Android system in wide use on mobile devices is built on the Linux kernel.

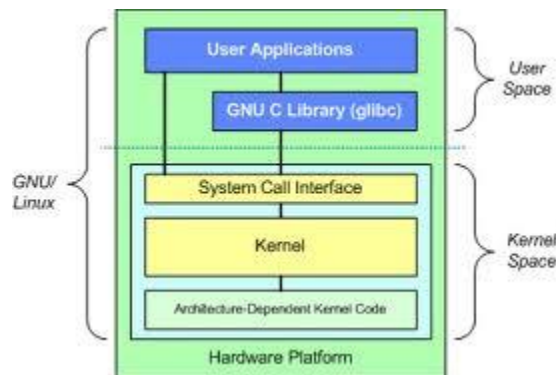


Fig. Linux Operating System

The development of Linux is one of the most prominent examples of free and open source software collaboration: the underlying source code may be used, modified, and distributed—commercially or non-commercially—by anyone under licenses such as the GNU General Public License. Typically Linux is packaged in a format known as a Linux distribution for desktop and server use. Some popular mainstream Linux distributions include Debian (and its derivatives such as Ubuntu), Fedora and openSUSE. Linux distributions include the Linux kernel, supporting utilities and libraries and usually a large amount of application software to fulfill the distribution's intended use.

Advantages of Linux

1. **Low cost:** You don't need to spend time and money to obtain licenses since Linux and much of its software come with the GNU General Public License. You can start to work immediately without worrying that your software may stop working anytime because the free trial version expires. Additionally, there are large repositories from which you can freely download high quality software for almost any task you can think of.
2. **Stability:** Linux doesn't need to be rebooted periodically to maintain performance levels. It doesn't freeze up or slow down over time due to memory leaks and such. Continuous up-times of hundreds of days (up to a year or more) are not uncommon.
3. **Performance:** Linux provides persistent high performance on workstations and on networks. It can handle unusually large numbers of users simultaneously, and can make old computers sufficiently responsive to be useful again.
4. **Network friendliness:** Linux was developed by a group of programmers over the Internet and has therefore strong support for network functionality; client and server

systems can be easily set up on any computer running Linux. It can perform tasks such as network backups faster and more reliably than alternative systems.

5. **Flexibility:** Linux can be used for high performance server applications, desktop applications, and embedded systems. You can save disk space by only installing the components needed for a particular use. You can restrict the use of specific computers by installing for example only selected office applications instead of the whole suite.
6. **Compatibility:** It runs all common Unix software packages and can process all common file formats.
7. **Choice:** The large number of Linux distributions gives you a choice. Each distribution is developed and supported by a different organization. You can pick the one you like best; the core functionalities are the same; most software runs on most distributions.
8. **Fast and easy installation:** Most Linux distributions come with user-friendly installation and setup programs. Popular Linux distributions come with tools that make installation of additional software very user friendly as well.
9. **Full use of hard disk:** Linux continues work well even when the hard disk is almost full.
10. **Multitasking:** Linux is designed to do many things at the same time, e.g., a large printing job in the background won't slow down your other work.
11. **Security:** Linux is one of the most secure operating systems. "Walls" and flexible file access permission systems prevent access by unwanted visitors or viruses. Linux users have to option to select and safely download software, free of charge, from online repositories containing thousands of high quality packages. No purchase transactions requiring credit card numbers or other sensitive personal information are necessary.
12. **Open Source:** If you develop software that requires knowledge or modification of the operating system code, Linux's source code is at your fingertips. Most Linux applications are Open Source as well.

Disadvantages of Linux:

1. **Understanding** – Becoming familiar with the Linux operating system requires patience as well as a strong learning curve. You must have the desire to read and figure things out on your own, rather than having everything done for you. Check out the [20 must read how-to's and guides for Linux](#).
2. **Compatibility** – Because of its free nature, Linux is sometimes behind the curve when it comes to brand new hardware compatibility. Though the kernel contributors and maintainers work hard at keeping the kernel up to date, Linux does not have as much of a corporate backing as alternative operating systems. Sometimes you can find third party applications, sometimes you can't.
3. **Alternative Programs** – Though Linux developers have done a great job at creating alternatives to popular Windows applications, there are still some applications that exist on Windows that have no equivalent Linux application.

Window XP

An operating system introduced in 2001 from Microsoft's Windows family of operating systems, the previous version of Windows being Windows Me. Microsoft called the release its most important product since Windows 95. Along with a redesigned look and feel to the user

interface, the new operating system is built on the Windows 2000 kernel, giving the user a more stable and reliable environment than previous versions of Windows. Windows XP comes in two versions, Home and Professional. The company has focused on mobility for both editions, including plug and play features for connecting to wireless networks. The operating system also utilizes the 802.11x wireless security standard.

Advantages of Window XP

1. It supports most of the hardware without need of external drivers.
2. It offers good GUI as compared to older versions.
3. It supports wireless networking
4. Many applications are developed for windows only and on other OS they do not work
5. Windows XP offers universal plug and play features
6. The availability of this is very high.
7. It offers universal solution to OS needs and there are no compatibility issues in this case.
8. Windows XP is user friendly as compared to other OS in market.

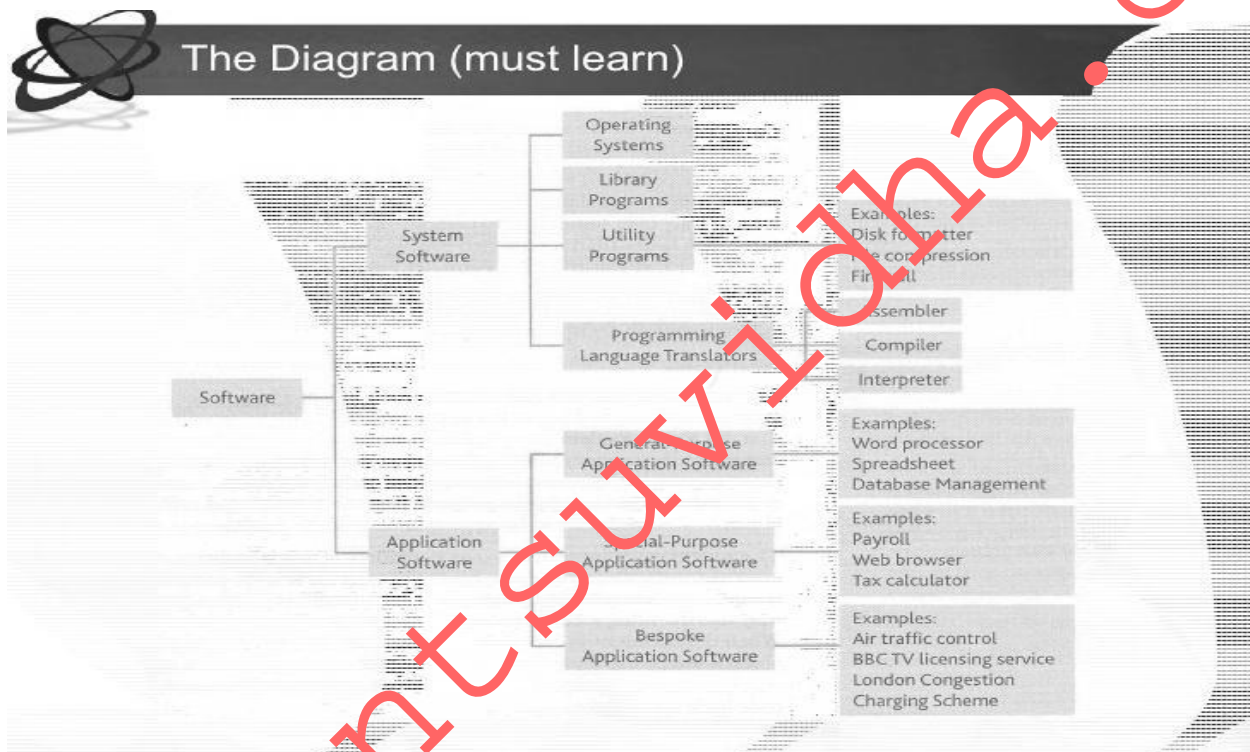
Disadvantages of Window XP

1. Security loopholes exist in it hence it is easily affected by virus and spyware.
2. It is expensive to purchase.
3. Prone to crashes and hence unstable.
4. Many flaws and bugs are present in it.
5. It is very heavy system and old hardware can't run it.
6. Older DOS programs may not run on this version.
7. Parallel port not recognized.
8. No file encryption facility in XP home edition.
9. Comes with single user license so it can't be loaded on multiple PCs.
10. It has no inbuilt chipset drivers making the install time consuming.

SECTION-B

Basic Introduction to Software

Classification of Software:-



Generations of Programming Language

There are many types of programming languages out there and you might already have heard of a few of them, for example: C++, VB.NET, HTML, Python, Assembly. We will now look at the history of how these languages came about and what they are still useful for. In all cases keep in mind that the only thing a computer will execute is machine code or **object code** when it has been converted from a language to run on a processor.

Generatio	First	Second	Third	Fourth
-----------	-------	--------	-------	--------

n				
Equivalent code	101010100110001 0 100110101000000 1 11111111010001 0	LDA 34 ADD #1 STO 34	x = x + 1	body.top { color : red; font-style : italic }
Language	(LOW) Machine Code	(LOW) Assembly Code	(HIGH) Visual Basic , C , python etc	(HIGH) SQL , CSS , Haskell etc
Relation to Object Code (generally)	--	one to one	one to many	one to many

First Generation Programming

The [first generation](#) program language is pure machine code, that is just ones and zeros, e.g.. Programmers have to design their code by hand then transfer it to a computer by using a punch card, punch tape or flicking switches. There is no need to translate the code and it will run straight away. This may sound rather archaic, but there are benefits:

- ⊕ Code can be fast and efficient
- ⊕ Code can make use of specific processor features such as special registers

And of course drawbacks

- ⊖ Code cannot be ported to other systems and has to be rewritten
- ⊖ Code is difficult to edit and update

Second generation programming

As you can imagine, writing in 1s and 0s all day will leave you prone to mistakes. [Second-generation](#) programming languages are a way of describing Assembly code which you may have already met.

By using codes resembling English programming becomes much easier. The use of these [mnemonic](#) codes such as LDA for load and STA for store means the code is easier to read and write. To convert an assembly code program into object code to run on a computer requires an **Assembler** and each line of assembly can be replaced by the equivalent one line of object (machine) code:

Assembly Code		Object Code
LDA A	-> Assembler ->	000100110100
ADD #5		001000000101
STA A		001100110100
JMP #3		010000000011

Assembly code has similar benefits to writing in machine code, it is a one to one relationship after all. This means that assembly code is often used when writing low level fast code for specific hardware. Until recently machine code was used to program things such as mobile phones, but with the speed and performance of languages such as [C](#) being very close to Assembly, and with C's ability to talk to processor registers, Assembly's use is declining.

As you can hopefully see there are benefits to using Second-Generation Languages over First-Generation, plus a few other things that makes Assembly great:

- + Code can be fast and efficient
- + Code can make use of specific processor features such as special registers
- + As it is closer to plain English, it is easier to read and write when compared to machine code

And of course drawbacks

- Code cannot be ported to other systems and has to be rewritten

Third generation (High Level Languages)

Even though Assembly code is easier to read than machine code, it is still not straightforward to perform loops and conditionals and writing large programs can be a slow process creating a mish-mash of [goto](#) statements and jumps. [Third-generation](#) programming languages brought many programmer-friendly features to code such as loops, conditionals, classes etc. This means that one line of third generation code can produce many lines of object (machine) code, saving a lot of time when writing programs.

Imperative languages - code is executed line by line, in sequence

Third generation (High Level Languages) codes are imperative. Imperative means that code is executed line by line, in sequence. For example:

```
1.  dim x as integer
2.
3.  x = 3
4.
5.  dim y as integer
6.
7.  y = 5
8.
9.  x = x + y
10.
11. console.writeline(x)
12.
```

Would output: 8

Third generation languages can be platform independent, meaning that code written for one system will work on another. To convert a 3rd generation program into object code requires a **Compiler** or an **Interpreter**, we'll look into these in more detail very soon.

To summarise:

- ➕ Hardware independence, can be easily ported to other systems and processors
- ➕ Time saving programmer friendly, one line of 3rd gen is the equivalent of many lines of 1st and 2nd gen

However

- ➖ Code produced might not make the best use of processor specific features unlike 1st and 2nd gen

Fourth generation

[Fourth-generation](#) languages are designed to reduce programming effort and the time it takes to develop software, resulting in a reduction in the cost of software development. They are not always successful in this task, sometimes resulting in inelegant and hard to maintain code.

Languages have been designed with a specific purpose in mind and this might include languages to query databases ([SQL](#)), languages to make reports ([Oracle Reports](#)) and languages to construct user interface ([XUL](#)). An example of 4th generation programming type is the declarative language

--an example of a Structured Query Language (SQL) to select criminal details from a database


```
SELECT name, height, DoB FROM criminals WHERE numScars = 7;
```

Declarative languages - describe what computation should be performed and not how to perform it. Not imperative!

An example of a declarative language is [CSS](#) which we'll learn more about when completing the web design unit

```
/*code to change the headings on a page to green and the paragraphs to red and italic*/  
h1 { color : green }  
p { color : red; font-style : italic }
```

Assembler

An **assembler** translates assembly language into machine code. Assembly language consists of mnemonics for machine opcodes so assemblers perform a 1:1 translation from mnemonic to a direct instruction. For example:

```
LDA #4 converts to 0001001000100100
```

Conversely, one instruction in a high level language will translate to one or more instructions at machine level.

Advantages of using an Assembler:

- + Very fast in translating assembly language to machine code as 1 to 1 relationship
- + Assembly code is often very efficient (and therefore fast) because it is a low level language
- + Assembly code is fairly easy to understand due to the use of English-like mnemonics

Disadvantages of using Assembler:

- Assembly language is written for a certain instruction set and/or processor
- Assembly tends to be optimised for the hardware it's designed for, meaning it is often incompatible with different hardware
- Lots of assembly code is needed to do relatively simple tasks, and complex programs require lots of programming time

Compiler

A **Compiler** is a computer program that **translates code** written in a high level language to a lower level language, object/machine code. The most common reason for translating source code is to create an executable program (converting from a high level language into machine language).

Advantages of using a compiler

- + Source code is not included, therefore compiled code is more secure than interpreted code
- + Tends to produce faster code than interpreting source code
- + Produces an executable file, and therefore the program can be run without need of the source code

Disadvantages of using a compiler

- Object code needs to be produced before a final executable file this can be a slow process
- The source code must be 100% correct for the executable file to be produced

Interpreter


An interpreter program executes other programs directly, running through program code and executing it line-by-line. As it analyses every line, an interpreter is slower than running compiled code but it can take less time to interpret program code than to compile and then run it — this is very useful when prototyping and testing code. Interpreters are written for multiple platforms, this means code written once can be run immediately on different systems without having to recompile for each. Examples of this include flash based web programs that will run on your PC, MAC, games console and Mobile phone.

Advantages of using an Interpreter

- + Easier to debug (check errors) than a compiler
- + Easier to create multi-platform code, as each different platform would have an interpreter to run the same code
- + Useful for prototyping software and testing basic program logic

Disadvantages of using an Interpreter

- Source code is required for the program to be executed, and this source code can be read making it insecure

 Interpreters are generally slower than compiled programs due to the per-line translation method

Loader

In [computing](#), a **loader** is the part of an [operating system](#) that is responsible for loading programs. It is one of the essential stages in the process of starting a program, as it places programs into memory and prepares them for execution. Loading a program involves reading the contents of [executable file](#), the file containing the program text, into memory, and then carrying out other required preparatory tasks to prepare the executable for running. Once loading is complete, the operating system starts the program by passing control to the loaded program code.

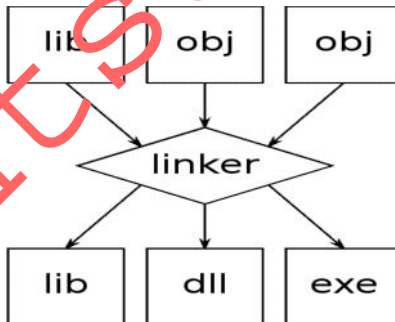
All operating systems that support program loading have loaders, apart from systems where code executes directly from ROM or in the case of highly specialized computer systems that only have a fixed set of specialised programs.

In many operating systems the loader is permanently resident in memory, although some operating systems that support [virtual memory](#) may allow the loader to be located in a region of memory that is [pageable](#).

Linker

In [computer science](#), a **linker** or **link editor** is a [program](#) that takes one or more [objects](#) generated by a [compiler](#) and combines them into a single [executable](#) program.

In [IBM mainframe](#) environments such as [OS/360](#) this program is known as a *linkage editor*.



The following Figure shows the steps involved in the process of building the C program starting from the compilation until the loading of the executable image into the memory for program running.

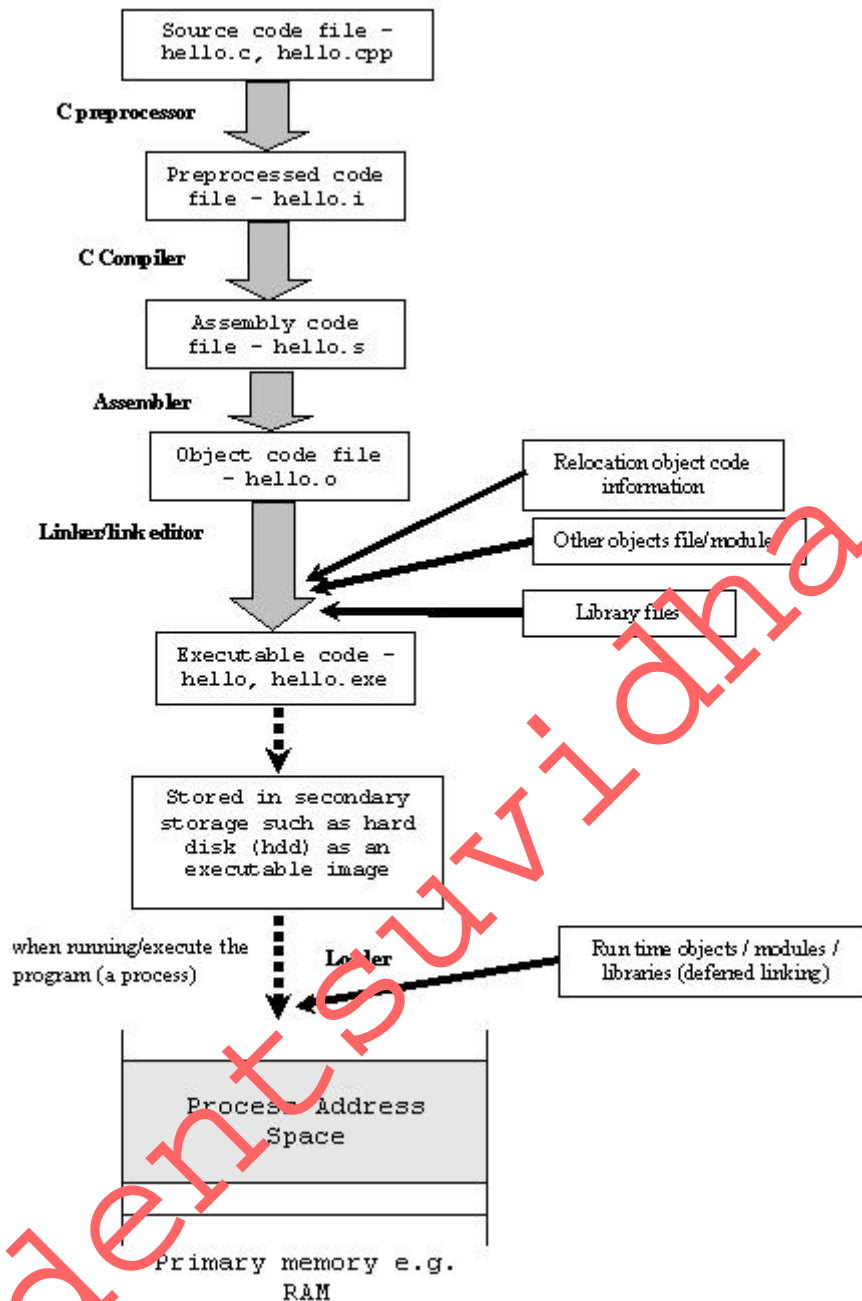


Figure w.1: Compile, link and execute stages for running program (a process)

Network

A network is a group of two or more computer systems linked together. There are many types of computer networks, including:

- **local-area networks (LANs)** : The computers are geographically close together (that is, in the same building).

- **wide-area networks (WANs)** : The computers are farther apart and are connected by telephone lines or radio waves.
- **campus-area networks (CANs)**: The computers are within a limited geographic area, such as a campus or military base.
- **metropolitan-area networks (MANs)**: A data network designed for a town or city.
- **home-area networks (HANs)**: A network contained within a user's home that connects a person's digital devices.

Data Communication

Data communication is the transmission of electronic data over some media. The media may be cables, microwaves.

Elements of Data Communication

Four basic elements are needed for any communication system.

1. **Sender.** The computer or device that is used for sending data is called sender, source or transmitter. In modern digital communication system, the source is usually a computer.
2. **Medium.** The means through which data is sent from one location to another is called transmission medium. If the receiver and transmitter are within a building, a wire connects them. If they are located at different locations, they may be connected by telephone lines, fiber optics or microwaves.
3. **Receiver.** The device or computer that receives the data is called receiver. The receiver can be a computer, printer or a fax machine.
4. **Protocols.** There are rules under which data transmission takes place between sender and receiver. The data communication s/w are used to transfer data from one computer to another. The s/w follows same communication protocols can communicate and exchange data.

Data Transmission

Data may be transfer from one device to another by means of some communication media. The electromagnetic or light waves that transfer data from one device to another device in encoded form are called signals. Data transmissions across the network can occur in two forms i.e.:

(i) **Analog signal.**

(ii) **Digital signal.**

Analog Signal. The transfer of data in the form of electrical signals or continuous waves is called analog signal or analog data transmission. An analog signal is measured in volts and its frequency is in hertz (Hz).

Advantages of Analog Signaling

- Allows multiple transmissions across the cable.
- Suffers less from attenuation.

Disadvantages of Analog Signaling

- Suffers from EMI.
- Can only be transmitted in one direction without sophisticated equipment.

Digital Signal

The transfer of data in the form of digit is called digital signal or digital data transmission. Digital signals consist of binary digits 0 & 1. Electrical pulses are used to represent binary digits. Data transmission between computers is in the form of digital signals.

Advantages of Digital Signaling

- Equipment is cheaper and simpler than analog equipment.
- Signals can be transmitted on a cable bidirectional.
- Digital signals suffer less from EMI.

Disadvantages Digital Signaling

- Only one signal can be sent at a time.
- Digital signals suffer from attenuation.

Techniques of Data Communication

There are two possible techniques of sending data from the sender to receiver, i.e.:-

(1) Parallel transmission.

(2) Serial transmission.

Parallel Transmission. In parallel transmission each bit of character / data has a separate channel and all bits of a character are transmitted simultaneously. Here the transmission is parallel character by character.

Sender Receiver

Serial Transmission. In serial transmission, the data is sent as one bit at a time having a signal channel for all the bit i.e.:-

Sender Receiver

Types of Serial Transmission

On serial transmission it is very essential to know exactly where one character ends and the next begins. The necessary synchronization that helps determine which bit is the first bit of the incoming character may be synchronous or asynchronous.

Asynchronous Serial Transmission

Computer communication that occurs one bit at a time with start and stop bits at the beginning and the end of each character is called Asynchronous Serial Transmission. In this type of transmission, there is no fixed time relationship with one character.

Advantages of Asynchronous Serial Transmission

- This type of transmission is very simple.
- This type of transmission is cheaper.

Disadvantages of Asynchronous Serial Transmission

- This type of transmission is slow.

Synchronous Transmission

In this method a clock signal is used and the sending as well as the receiving devices are synchronized with this clock signals. It doesn't use start and stop bits but the character are sent in character groups called block

Advantages of Synchronous Transmission

- It is very fast as compared to Asynchronous Series Transmission.

Disadvantage of Synchronous Transmission

- It uses more expensive and complex equipment.

Modes of Data Communication.

The manner in which data is transmitted from one location to another location is called data transmission mode.

There are three ways or modes for transmitting data from one location to another. These are:

- (1) **Simplex.**
- (2) **Half duplex.**
- (3) **Full duplex.**

Half Duplex.

In half duplex mode data can be transmitted in both directions but only in one direction at a time. During any transmission, one is the transmitter and the other is receiver. So each time for sending or receiving data, direction of data communication is reversed, this slow down data transmission rate. In half duplex modes, transmission of data can be confirmed.

Half Duplex Mode

Wireless communication is an example of half duplex.

Advantages of Half Duplex

- Costs less than full duplex.
- Enables for two way communications.

Disadvantages of Half Duplex

- Costs more than simplex.
- Only one device can transmit at a time.

Simplex Mode

In simplex mode, data is transmitted in only one direction. A terminal can only send data and cannot receive it or it can only receive data but cannot send it. Simplex mode is usually used for a remote device that is meant only to receive data. It is not possible to confirm successful transmission of data in simplex mode.

This mode is not widely used. Speaker, radio and television broadcasting are examples of simplex transmission, on which the signal is sent from the transmission to your TV antenna. There is no return signal.

Simplex Mode

Advantages of Simplex

- Cheapest communication method.

Disadvantage of Simplex

- Only allows for communication in one direction.

Full Duplex.

In full mode, data can be transmitted in both directions simultaneously. It is a faster mode for transmitting data because no time wastes in switching directions.

Full Duplex Mode

Example of full duplex is telephone set in which both the users can talk and listen at the same time.

Advantage of Full Duplex.

- Enables two-way communication simultaneously.

Disadvantage of Full Duplex.

The most expensive method in terms of equipment because of two bandwidth channels is required.

TOPOLOGY

A network topology refers to the way in which nodes in a network are connected to one another. The way in which they are connected defines how they communicate. Network topologies describe the ways in which the elements of a network are connected. They describe the physical and logical arrangement of network nodes.

There are six types of topologies:

- i. Mesh topology.
- ii. Star topology.
- iii. Ring topology.
- iv. Tree topology.
- v. Bus topology.
- vi. Hybrid topology.

Mesh Topology:

In a mesh topology, every device has a dedicated point-to-point link to every other device. The term dedicated means that the link carries traffic only between the two devices it connects. Here, if we have n nodes, then we need to connect to $n-1$ nodes and $n(n-1)$ physical links. However, if each physical link allows communication in both directions (duplex mode), we need $n(n-1)/2$ links.

Advantages of a Mesh Topology

- Eliminates traffic problems in links sharing.
- If one link becomes unusable, it does not incapacitate the entire system. Thus, act as robust.
- It has privacy and security.
- Point-to-point link make fault identification and fault isolation easy.

Disadvantages of a Mesh Topology

- Installation and reconnection are difficult.
- The hardware required to connect each link (I/O ports and cable) is expensive.
- It is generally too costly and complex for practical networks.

Star Topology:

In local area networks where the star topology is used, each machine is connected to a central hub. The star topology allows each machine on the network to have a point to point connection to the central hub. All of the traffic which transverses the network passes through the central hub. The hub acts as a signal booster or repeater which in turn allows the signal to travel greater distances.

Advantages of a Star Topology

- Easy to install and reconfigure.
- No disruptions to the network when connecting or removing devices.
- Easy to detect faults and to remove parts.
- Less expensive.

- Includes robustness, that is, if one link fails, only that link is affected, other links remain active.

Disadvantages of a Star Topology

- If the hub fails, the whole system is dead.
- If the hub, switch, or concentrator fails, nodes attached are disabled.
- Requires more cable length than a bus topology.
- More expensive than bus topologies because of the cost of the hubs, etc.

Ring Topology:

In local area networks where the ring topology is used, each computer is connected to the network in a closed loop or ring. The signal passes through each machine or computer connected to the ring in one direction, from device to device, until it reaches its destination. Each machines or computers connected to the ring act as signal boosters or repeaters. When a device receives a signal intended for another device, its repeater regenerates the bits and passes them along.

Advantages of a Ring Topology

- It is relatively easy to install and reconfigure.
- Easy to identify the problem if the entire network shuts down.

Disadvantages of a Ring Topology

- Only one machine can transmit on the network at a time.
- The failure of one machine will cause the entire network to fail.

Tree Topology:

The type of network topology in which a central 'root' node (the top level of the hierarchy) is connected to one or more other nodes that are one level lower in the hierarchy (i.e., the second level) with a point-to-point link between each of the second level nodes and the top level central 'root' node, while each of the second level nodes that are connected to the top level central 'root' node will also have one or more other nodes that are one level lower in the hierarchy (i.e., the third level) connected to it, also with a point-to-point link, the top level central 'root' node being the only node that has no other node above it in the hierarchy.

Advantages of a Tree Topology

- Point-to-point wiring for individual segments.
- Supported by several hardware and software vendors.

Disadvantages of a Tree Topology

- Overall length of each segment is limited by the type of cabling used.
- If the backbone line breaks, the entire segment goes down.
- More difficult to configure and wire than other topologies.

Bus Topology:

In local area networks where bus technology is used, each machine is connected to a long, single cable. The cable acts as a backbone to link all the devices in a network. Each

computer or server is connected to the single bus cable through drop lines and some kind of connector. A terminator is required at each end of the bus cable to prevent the signal from bouncing back and forth on the bus cable.

Advantages of Bus Topology

- Easy to connect a computer or peripheral to a linear bus.
- Requires less cable length than mesh or star topologies.
- It is cheaper than any other topologies.
-

Disadvantages of Bus Topology

- If the network cable breaks, the entire network will be down.
- Terminators are required at both ends of the backbone cable.
- Difficult to identify the problem if the entire network shuts down.
- Not meant to be used as a stand-alone solution in a large building.
- Include difficult reconnection and fault isolation.
- The managing cost of network is too high.
- Addition of new devices requires modification or replacement of the backbone.

Hybrid Topology:

Hybrid networks use a combination of any two or more topologies in such a way that the resulting network does not exhibit one of the standard topologies (e.g., bus, star, ring, etc.). A hybrid topology is always produced when two different basic network topologies are connected.

Advantages of a Hybrid Topology

- It provides a better result by it.
- It can be designed in many ways for various purposes.

Disadvantages of Hybrid Topology

- It is costly.
- Difficult to identify the problem if the entire network shuts down.

OSI Reference Model

Open Systems Interconnection ([OSI](#)) is a standard reference model for communication between two end users in a network. The model is used in developing products and understanding networks.

OSI divides telecommunication into seven layers. The layers are in two groups. The upper four layers are used whenever a message passes from or to a user. The lower three layers are used when any message passes through the host computer. Messages intended for this computer pass to the upper layers. Messages destined for some other host are not passed up to the upper layers but are forwarded to another host. The seven layers are:

Layer 7: The application layer ...This is the layer at which communication partners are identified, quality of service is identified, user authentication and privacy are considered, and any

constraints on data syntax are identified. (This layer is *not* the application itself, although some applications may perform application layer functions.)

Layer 6: The presentation layer ...This is a layer, usually part of an operating system, that converts incoming and outgoing data from one presentation format to another (for example, from a text stream into a popup window with the newly arrived text). Sometimes called the syntax layer.

Layer 5: The session layer ...This layer sets up, coordinates, and terminates conversations, exchanges, and dialogs between the applications at each end. It deals with session and connection coordination.

Layer 4: The transport layer ...This layer manages the end-to-end control (for example, determining whether all packets have arrived) and error-checking. It ensures complete data transfer.

Layer 3: The network layer ...This layer handles the routing of the data (sending it in the right direction to the right destination on outgoing transmissions and receiving incoming transmissions at the packet level). The network layer does routing and forwarding.

Layer 2: The data-link layer ...This layer provides synchronization for the physical level and does bit-stuffing for strings of 1's in excess of 5. It furnishes transmission protocol knowledge and management.

Layer 1: The physical layer ...This layer conveys the bit stream through the network at the electrical and mechanical level. It provides the hardware means of sending and receiving data on a carrier.

Network Security:-

The networks are computer networks, both public and private, that are used every day to conduct transactions and communications among businesses, government agencies and individuals. The networks are comprised of "nodes", which are "client" terminals (individual user PCs) and one or more "servers" and/or "host" computers. They are linked by communication systems, some of which might be private, such as within a company, and others which might be open to public access. The obvious example of a network system that is open to public access is the Internet, but many private networks also utilize publicly-accessible communications. Today, most companies' host computer can be accessed by their employees whether in their offices over a private communications network, or from their homes or hotel rooms while on the road through normal telephone lines.

Network security involves all activities that organizations, enterprises, and institutions undertake to protect the value and ongoing usability of assets and the integrity and continuity of operations. An effective network security strategy requires identifying threats and then choosing the most effective set of tools to combat them.

Threats to network security include:

Viruses : Computer programs written by devious programmers and designed to replicate themselves and infect computers when triggered by a specific event

Trojan horse programs : Delivery vehicles for destructive code, which appear to be harmless or useful software programs such as games

Vandals : Software applications or applets that cause destruction

Attacks : Including reconnaissance attacks (information-gathering activities to collect data that is later used to compromise networks); access attacks (which exploit network vulnerabilities in order to gain entry to e-mail, databases, or the corporate network); and denial-of-service attacks (which prevent access to part or all of a computer system)

Data interception : Involves eavesdropping on communications or altering data packets being transmitted

Social engineering : Obtaining confidential network security information through nontechnical means, such as posing as a technical support person and asking for people's passwords

Network security tools include:

Antivirus software packages : These packages counter most virus threats if regularly updated and correctly maintained.

Secure network infrastructure : Switches and routers have hardware and software features that support secure connectivity, perimeter security, intrusion protection, identity services, and security management.

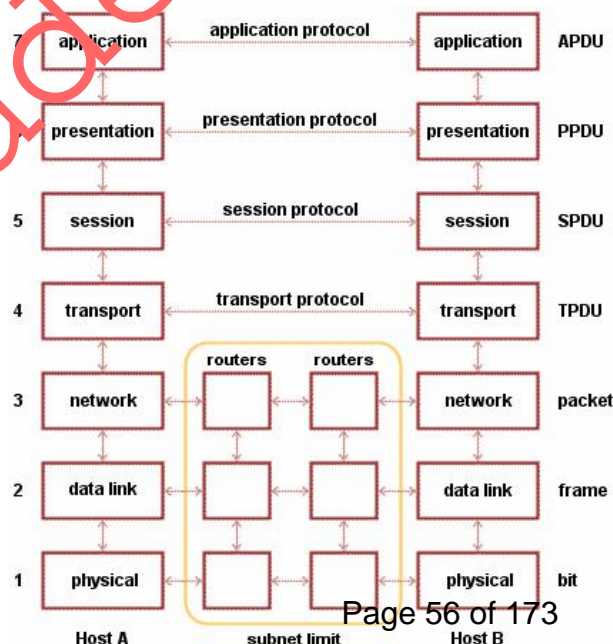
Dedicated network security hardware and software—Tools such as firewalls and intrusion detection systems provide protection for all areas of the network and enable secure connections.

Virtual private networks : These networks provide access control and data encryption between two different computers on a network. This allows remote workers to connect to the network without the risk of a hacker or thief intercepting data.

Identity services : These services help to identify users and control their activities and transactions on the network. Services include passwords, digital certificates, and digital authentication keys.

Encryption : Encryption ensures that messages cannot be intercepted or read by anyone other than the authorized recipient.

Security management : This is the glue that holds together the other building blocks of a strong security solution.



Types of Networks.(LAN, WAN, MAN).

	LAN	MAN	WAN
1.	Local	Metropolitan	Wide
2.	Used guided media	guided or unguided	Unguided
3.	A communication network linking a number of stations in some local area.	This network shares the characteristics of packet broad casting networks.	A communication network distinguished from a local area network
4.	Serve in a limited boundary	Large geographical area	Long distance communication.
5.	Speed high	Speed Less then LAN	Speed Slower than LAN
6.	LAN generally provides a high-speed 100 Kbps to 100 Mbps.	A MAN is optimized for a large geographical area than LAN.	Is long distance communications, which may or may not be provided by public packet network.
7.	Expense low	Expense high	Expense high
8.	Used by office, home, school.	Used ATM.	Used Govt bodies large org.

The TCP/IP reference model

The TCP/IP model

TCP/IP is based on a four-layer reference model. All protocols that belong to the TCP/IP protocol suite are located in the top three layers of this model.

As shown in the following illustration, each layer of the TCP/IP model corresponds to one or more layers of the seven-layer Open Systems Interconnection (OSI) reference model proposed by the International Standards Organization (ISO).

TCP/IP Layers	TCP/IP Protocols				
Application Layer	HTTP	FTP	Telnet	SMTP	DNS
Transport Layer	TCP		UDP		
Network Layer	IP	ARP	ICMP	IGMP	
Network Interface Layer	Ethernet	Token Ring		Other Link-Layer Protocols	

The types of services performed and protocols used at each layer within the TCP/IP model are described in more detail in the following table.

Layer	Description	Protocols
Application	Defines TCP/IP application protocols and how host programs interface with transport layer services to use the network.	HTTP, Telnet, FTP, TFTP, SNMP, DNS, SMTP, X Windows, other application protocols
Transport	Provides communication session management between host computers. Defines the level of service and status of the connection used when transporting data.	TCP, UDP, RTP
Internet	Packages data into IP datagrams, which contain source and destination address information that is used to forward the datagrams between hosts and across networks. Performs routing of IP datagrams.	IP, ICMP, ARP, RARP
Network interface	Specifies details of how data is physically sent through the network, including how bits are electrically signaled by hardware devices that interface directly with a network medium, such as coaxial cable, optical fiber, or twisted-pair copper wire.	Ethernet, Token Ring, FDDI, X.25, Frame Relay, RS-232, v.35

Network Connecting Devices

Various devices are used to connect network of a computer. The most common devices are:

- 1) **Routers**
- 2) **Gateways**
- 3) **Repeaters**
- 4) **Bridges**
- 5) **Hub**
- 6) **Modem**

ROUTER

Routers are devices which connect two or more networks that use similar protocol. A router consists of hardware and software.

Hardware can be a computer or a specific device.

Software consists of a special management program that controls flow of data between networks.

Routers operate at a network layer of O.S.I model.

Routers use logical and physical address to connect two or more logically separate networks. They make this connection by organizing the large network into logical network segments (sometimes small sub-networks or sub-nets). Each of these sub-nets is given a logical address. Data is grouped into packets or blocks of data.

Each packet in addition to having a physical device address, has a logical address. The network address allows routers to calculate more accurately and efficiently the path of the computer.

Advantages of Router

They use a high level of intelligence to route data.

Routers can also act as a bridge to handle non-routable protocols such as NetBEUI (Network Bios Extended User Interface).

Disadvantages of Router:

- High level of intelligence takes more processing time which can affect performance.
- Routers are very complicated which installation and maintenance are difficult.

GATEWAYS

Gateways are devices which connect two or more networks that use different protocols. They are similar in function to routers but they are more powerful and intelligent devices. A gateway can actually convert data so that network with an application on a computer on the other side of the gateway e.g. a gateway can receive email messages in one format and convert them into another.

format. Gateway can operate at all seven layer of OSI model. Since Gateway perform data conversion so they are slower in speed and very expensive devices.

REPEATERS

Repeaters are used within network to extend the length of communication.

Data process through transmission media in the form of waves or signals. The transmission media weaken signals that move through it. The weakening of signal is called attenuation. If the data is to be transmitted beyond the maximum length of a communication media, signals have to be amplified. The devices that are used to amplify the signals are called repeaters. Repeaters work at the physical layer of OSI model.

Repeaters are normally two ports boxes that connect two segments. As a signal comes in one port, it is Regenerated and send out to the other port.

The signal is read as 1s and 0s. As 1s and 0s are transmitted, the noise can be cleaned out.

Advantages of Repeater

- Repeaters easily extend the length of network.
- They require no processing overhead, so very little if any performance degradation occurs.
- It can connect signals from the same network type that use different types of cables.

Disadvantages of Repeaters

- Repeaters can not be used to connect segments of different network types.
- They cannot be used to segment traffic on a network to reduce congestion.
- Many types of network have a limit on the number of network s that can be used at once.

BRIDGES

Bridges are used to connect similar network segments.

A bridge does not pass on signals it receives. When a bridge receive a signal, it determines its destination by looking at its destination and it sends the signals towards it. For example in a above figure a bridge has been used to join two network segments A AND B.

When the bridge receives the signals it read address of both sender and receiver. If the sender is a computer in segment A and the receiver is also segment A, it would not pass the signals to the segments B. It will however pass signals if the sender is in one segment and the receiver in other segment. Bridge works at the data link layer of O.S.I model.

Advantages of Bridges

- Bridge extends network segments by connecting them together to make one logical network.

- They can affect the segment traffic between networks by filtering data if it does not need to pass.
- Like repeaters they can connect similar network types with different cabling.

Disadvantages of Bridges

- Bridge possess information about the data they receive with can slow performance.

HUB

Hubs are basically multi ports repeaters for U.T.P cables. Some hubs have ports for other type of cable such as coaxial cable. Hubs range in size from four ports up to and for specific to the network types. These are some hubs which are

I. Passive Hub

II. Active Hub

III. Switch/ Intelligent Hub

Passive Hub

It provides no signal regeneration. They are simply cables connected together so that the signal is broken out to other nodes with out regeneration. These are not used often today because of loss of cable length that is allowed.

Active Hub

It acts as repeaters and regenerates the data signals to all ports. They have no real intelligence to tell weather the signal needs to go to all ports that is blindly repeated.

Switch Hub

Switches are multi ports bridges. They filter traffic between the ports on the switch by using the address of computers transmitting to them.

Switches can be used when data performance is needed or when collision need to be reduce.

Advantages of Hub

- Hubs need almost no configuration.
- Active hub can extend maximum network media distance.

No processing is done at the hub to slow down performance

Disadvantages of Hub

- Passive hubs can greatly limit maximum media distance.
- Hubs have no intelligence to filter traffic so all data is sent out on all ports whether it is needed or not.

Since hubs can act as repeaters the network using them must follow the same rules as repeaters.

MODEM

The device that converts digital signals into analog signals and analog signals to digital signals is called Modem. The word modem stands for modulation and demodulation. The process of converting digital signals to analog signals is called modulation. The process of converting analog signals to digital signals is called demodulation. Modems are used with computers to transfer data from one computer to another computer through telephone lines.

Modems have two connections these are.

- **Analog connection**
- **Digital connection**

Analog connection.

The connection between the modem and the telephone line is called analog connection.

Types of Modem:-there are two types of modems

- **Internal modem**
- **External modem**

Digital connection.

The connection of modem to computer is called digital connection.

INTERNAL MODEM

It fits into expansion slots inside the computer. It is directly linked to the telephone lines through the telephone jack. It is normally less expensive than external modem. Its transmission speed is also less than external modem.

EXTERNAL MODEM

It is an external unit of computer and is connected to the computer through serial port. It is also linked to the telephone line through a telephone jack. External modems are expensive and have more operation features and high transmission speed.

Advantages of Modem

- i. Inexpensive hardware and telephone lines.
- ii. Easy to setup and maintain.

Disadvantages of Modem

i. Very slow performance.

Domain Name Server

If you've ever used the [Internet](#), it's a good bet that you've used the **Domain Name System**, or **DNS**, even without realizing it. DNS is a protocol within the set of standards for how computers exchange data on the Internet and on many private networks, known as the TCP/IP protocol suite. Its basic job is to turn a user-friendly **domain name** like "howstuffworks.com" into an [Internet Protocol \(IP\) address](#) like 70.42.251.42 that computers use to identify each other on the network. It's like your computer's GPS for the Internet.

Computers and other network devices on the Internet use an IP address to route your request to the site you're trying to reach. This is similar to dialing a phone number to connect to the person you're trying to call. Thanks to DNS, though, you don't have to keep your own address book of IP addresses. Instead, you just connect through a **domain name server**, also called a **DNS server** or **name server**, which manages a massive database that maps domain names to IP addresses.

Whether you're accessing a Web site or sending [e-mail](#), your computer uses a DNS server to look up the domain name you're trying to access. The proper term for this process is **DNS name resolution**, and you would say that the DNS server resolves the domain name to the IP address. For example, when you enter "http://www.howstuffworks.com" in your browser, part of the network connection includes resolving the domain name "howstuffworks.com" into an IP address, like 70.42.251.42, for HowStuffWorks' Web servers.

You can always bypass a DNS lookup by entering 70.42.251.42 directly in your browser (give it a try). However, you're probably more likely to remember "howstuffworks.com" when you want to return later. In addition, a Web site's IP address can change over time, and some sites associate multiple IP addresses with a single domain name.

Without DNS servers, the Internet would shut down very quickly. But how does your computer know what DNS server to use? Typically, when you connect to your [home network](#), Internet service provider (ISP) or WiFi network, the modem or router that assigns your computer's network address also sends some important network configuration information to your computer or mobile device. That configuration includes one or more DNS servers that the device should use when translating DNS names to IP address.

So far, you've read about some important DNS basics. The rest of this article dives deeper into domain name servers and name resolution. It even includes an introduction to managing your own DNS server. Let's start by looking at how IP addresses are structured and how that's important to the name resolution process.

HTTP:

The Hypertext Transfer Protocol (HTTP) is an application-level TCP/IP based protocol with the lightness and speed necessary for distributed, collaborative, hypermedia information systems (internet).

HTTP stands for Hypertext Transfer Protocol

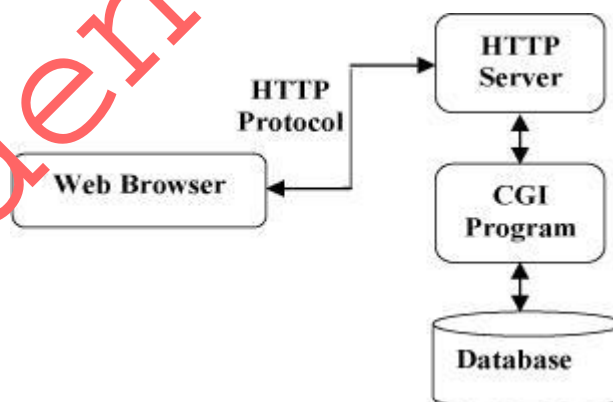
HTTP stands for **Hypertext Transfer Protocol**. It is an TCP/IP based communication protocol which is used to deliver virtually all files and other data, collectively called resources, on the World Wide Web. These resources could be HTML files, image files, query results, or anything else.

A browser works as an HTTP client because it sends requests to an HTTP server, which is called Web server. The Web Server then sends responses back to the client. The standard and default port for HTTP servers to listen on is 80 but it can be changed to any other port like 8080 etc.

There are three important things about HTTP of which you should be aware:

- **HTTP is connectionless:** After a request is made, the client disconnects from the server and waits for a response. The server must re-establish the connection after it processes the request.
- **HTTP is media independent:** Any type of data can be sent by HTTP as long as both the client and server know how to handle the data content. How content is handled is determined by the MIME specification.
- **HTTP is stateless:** This is a direct result of HTTP's being connectionless. The server and client are aware of each other only during a request. Afterwards, each forgets the other. For this reason neither the client nor the browser can retain information between different requests across the web pages.

Following diagram shows where HTTP Protocol fits in communication:



SECTION-C

Introduction to C

History

The C language was developed at AT&T Bell Labs in the early 1970s by Dennis Ritchie. It was based on an earlier Bell Labs language "B" which itself was based on the BCPL language. Since early on, C has been used with the Unix operating system, but it is not bound to any particular O/S or hardware.

C has gone through some revisions since its introduction. The American National Standards Institute developed the first standardized specification for the language in 1989, commonly referred to as C89. Before that, the only specification was an informal one from the book "The C Programming Language" by Brian Kernighan and Dennis Ritchie.

The next major revision was published in 1999. This revision introduced some new features, data types and some other changes. This is referred to as the C99 standard.

Advantages

Here are some advantages of programming in C:

- C is a general purpose programming language, meaning that it is not limited to any one specific kind of programming. This is different from languages like COBOL which was built for business applications, and FORTRAN for scientific calculations. You can write all sorts of software using C.
- C is not a very high-level language. A high-level language tries to isolate the programmer from the hardware as much as possible. In contrast, C allows you to directly access memory addresses, create bit fields and structures and map them to memory, perform bitwise operations and so on. C facilitates hardware programming.
- Not being high level also means there is little overhead; it is highly efficient and provides fast execution speed.
- There are C language compilers and development tools available for many different platforms from small embedded systems to large mainframes and supercomputers.
- C has been around for almost 40 years. In that time there has much software written in C. If there is some functionality you need in a C program you are writing, chances are someone has already written it. It may even be available for free.

Uses of C

In spite of its age, C is still being heavily used in industry. Several surveys have placed C as one of the most popular languages currently in use.

C is a very good choice for writing software to control hardware. The Unix (and derivatives) operating system's kernel is written in C (with some small pieces in assembly). Most firmware and device drivers are written in C as well.

C is also used in many real-time systems programming. While the language itself does not have any real-time features, it can be combined with platform-specific libraries or libraries that implement the POSIX real-time interfaces. C is a very efficient language that does not require many supporting libraries to run and does not have much overhead, which is desirable in low-memory embedded systems. Combining real-time libraries with C give it the timing constraints and other features needed for real-time programming.

Because C is efficient and fast it is sometimes used as the development language of other programming languages. Languages like PHP and Perl have been written in C. Many computationally intensive libraries and applications like MATLAB have been written in it too, for the same reason.

We have only talked about a few specialized domains where C is used. In addition to those, there are many other applications of all kinds that are written in C.

Structure of C Program

In this section we will take a look at the structure of a C program. Remember that many of the concepts, terms and syntax shown in this section will be reviewed in detail in other tutorials. This is only an introduction.

A C program may be made up of one or more files called "source files". There is a kind of source file that is used to define constants, macros, function prototypes, type definitions, etc. called a "header file". Header files are basically used to share things between other source files. By convention, source file names have the extension ".c" and header file names have the extension ".h".

How you enter a C statements into one or more files, how you run the compiler on those files, and how you run the resulting executable is completely dependent on what system you are using and what tools you have. Most systems have some kind of text editor for creating and modifying files.

Each compiler is different; you must consult your compiler's documentation for information on how to run it and how to set different options. There are also Integrated Development Environments (IDEs) that let you edit, compile, run and sometimes debug a program, all with a friendly user interface. The examples in this tutorial were written using a text editor on a Linux system and compiled with the gcc compiler.

Let us look at a very basic C program. We will write the canonical "Hello World" program in a file called hello.c. Here are the contents of file hello.c:

Sample Code

```

1. #include <stdio.h>
2.
3. int main(void)
4. {
5.     printf("Hello Worldn");
6.     return 0;
7. }

```

The C Character Set

A character denotes any alphabet, digit or special symbol used to represent information. Figure 1 shows the valid alphabets, numbers and special symbols allowed in C.

Alphabets	A, B,, Y, Z a, b,, y, z
Digits	0, 1, 2, 3, 4, 5, 6, 7, 8, 9
Special symbols	~ ' ! @ # % ^ & * () _ + = \ { } [] : ; " ' < > , . ? /

Figure 1

Constants, Variables and Keywords

The alphabets, numbers and special symbols when properly combined form constants, variables and keywords. Let us see what are 'constants' and 'variables' in C. A constant is an entity that doesn't change whereas a variable is an entity that may change.

In any program we typically do lots of calculations. The results of these calculations are stored in computers memory. Like human memory the computer memory also consists of millions of cells. The calculated values are stored in these memory cells. To make the retrieval and usage of these values easy these memory cells (also called memory locations) are given names. Since the value stored in each location may change the names given to these locations are called variable names. Consider the following example.

Here 3 is stored in a memory location and a name **x** is given to it. Then we are assigning a new value 5 to the same memory location **x**. This would overwrite the earlier value 3, since a memory location can hold only one value at a time. This is shown in Figure 2.

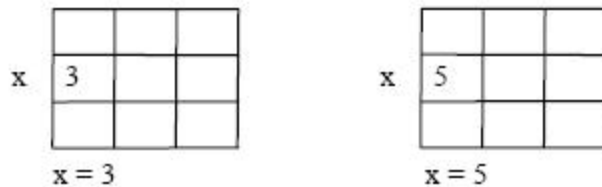


Figure 2

Types of C Constants

C constants can be divided into two major categories:

1. Primary Constants
2. Secondary Constants

These constants are further categorized as shown in Figure 3

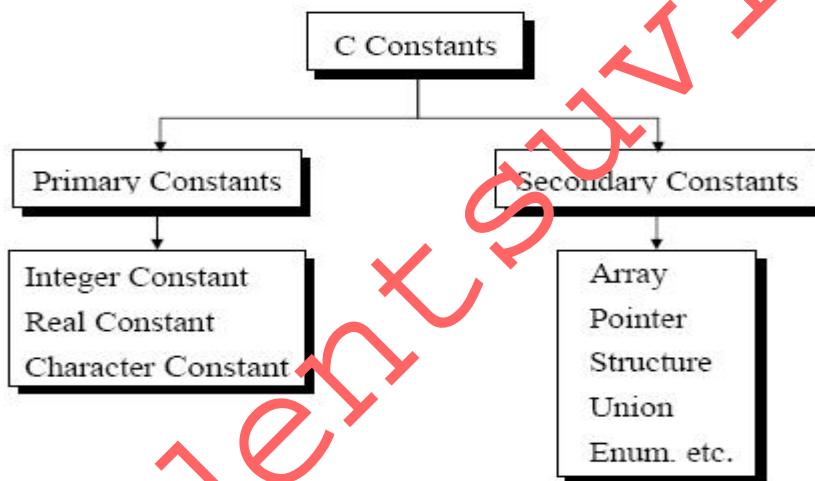


Figure 3

Types of C Variables

As we saw earlier, an entity that may vary during program execution is called a variable.

Variable names are names given to locations in memory. These locations can contain integer, real or character constants. In any language, the types of variables that it can support depend on the types of constants that it can handle. This is because a particular type of variable can hold only the same type of constant. For example, an integer variable can hold only an integer

constant, a real variable can hold only a real constant and a character variable can hold only a character constant.

C Keywords

Keywords are the words whose meaning has already been explained to the C compiler (or in a broad sense to the computer). The keywords **cannot** be used as variable names because if we do so we are trying to assign a new meaning to the keyword, which is not allowed by the computer. Some C compilers allow you to construct variable names that exactly resemble the keywords. However, it would be safer not to mix up the variable names and the keywords. The keywords are also called 'Reserved words'.

There are only 32 keywords available in C. Figure 4 gives a list of these keywords for your ready reference.

auto	double	int	struct
break	else	long	switch
case	enum	register	typedef
char	extern	return	union
const	float	short	unsigned
continue	for	signed	void
default	goto	sizeof	volatile
do	if	static	while

Figure 4

Data Types in C Language

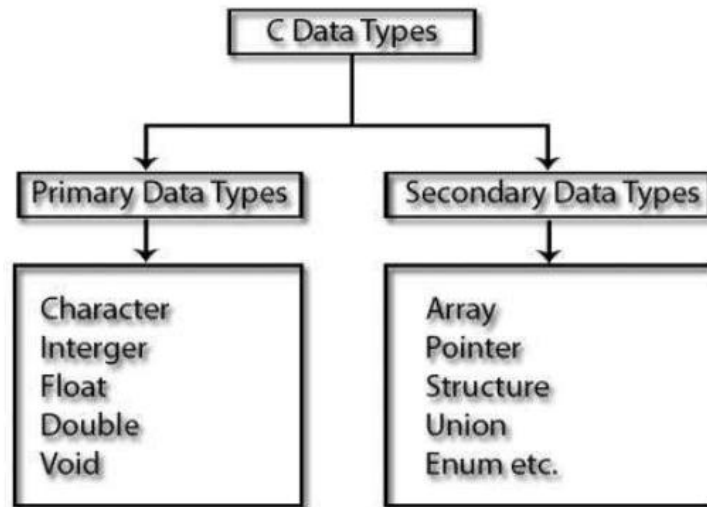
A programming language is proposed to help programmer to process certain kinds of data and to provide useful output. The task of data processing is accomplished by executing series of commands

called program. A program usually contains different types of data types (integer, float, character etc.) and need to store the values being used in the program. C language is rich of data types. A C

programmer has to employ proper data type as per his requirements.

C has different data types for different types of data and can be broadly classified as:

1. Primary Data Types
2. Secondary Data Types



Primary Data Types:

Integer Data Types:

Integers are whole numbers with a range of values, range of values are machine dependent. Generally an integer occupies 2 bytes memory space and its value range limited to -32768 to +32767

(that is, -215 to +215-1). A signed integer use one bit for storing sign and rest 15 bits for number. To control the range of numbers and storage space, C has three classes of integer storage namely short int, int and long int. All three data types have signed and unsigned forms. A short int requires half the amount of storage than normal integer. Unlike signed integer, unsigned integers are always positive and use all the bits for the magnitude of the number. Therefore, the range of an unsigned integer will be from 0 to 65535. The long integers are used to declare a longer

range of values and it occupies 4 bytes of storage space.

Syntax:

```

int <variable name>;
int num1;
short int num2;
long int num3;
  
```

Example: 5, 6, 100, 2500.

Integer Data Type Memory Allocation:

short int	int	long int
1 Byte	2 Bytes	4 Bytes

Floating Point Data Types:

The float data type is used to store fractional numbers (real numbers) with 6 digits of precision. Floating point numbers are denoted by the keyword `float`. When the accuracy of the floating point

number is insufficient, we can use the `double` to define the number. The double is same as `float` but with longer precision and takes double space (8 bytes) than `float`. To extend the precision

further we can use `long double` which occupies 10 bytes of memory space.

Syntax:

```
float <variable name>;  
float num1;  
double num2;  
long double num3;
```

Example: 9.125, 3.1254.

Floating Point Data Type Memory Allocation:

float	double	long double
4 Bytes	8 Bytes	10 Bytes

Character Data Type:

Character type variable can hold a single character and are declared by using the keyword `char`. As

there are signed and unsigned `int` (either short or long), in the same way there are signed and unsigned `chars`; both occupy 1 byte each, but having different ranges. Unsigned characters have values between 0 and 255, signed characters have values from -128 to 127.

Syntax:

```
char <variable name>;  
char ch = 'a';
```

Example: a, b, g, S, j.

Void Type:

The `void` type has no values therefore we cannot declare it as variable as we did in case of integer

and float. The `void` data type is usually used with function to specify its type.

Operators and Expressions

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:

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 denominator	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	!(A && B) is false.

	condition is true then Logical NOT operator will make false.	
--	--	--

Bitwise Operators:

Bitwise operator works on bits and perform 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 effect 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	A << 2 will give 240 which is 1111 0000

	specified by the right operand.	
>>	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 assign 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	C /= A is equivalent to C = C / A

	operand	
%=	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.	sizeof(a), where a is interger, 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 categorised 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 precedence than + so it first gets 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

Managing Input/ Output Operations

Types of I/O

Though C has no provision for I/O, it of course has to be dealt with at some point or the other. There is not much use writing a program that spends all its time telling itself a secret. Each Operating System has its own facility for inputting and outputting data from and to the files and devices. It's a simple matter for a system programmer to write a few small programs that would link the C compiler for particular Operating system's I/O facilities.

The developers of C Compilers do just that. They write several standard I/O functions and put them in libraries. These libraries are available with all C compilers. Whichever C compiler you are using it's almost certain that you have access to a library of I/O functions.

Do understand that the I/O facilities with different operating systems would be different. Thus, the way one OS displays output on screen may be different than the way another OS does it. For example, the standard library function **printf()** for DOS-based C compiler has been written keeping in mind the way DOS outputs characters to screen. Similarly, the **printf()** function for a Unix-based compiler has been written keeping in mind the way Unix outputs characters to screen. We as programmers do not have to bother about which **printf()** has been written in what manner. We should just use **printf()** and it would take care of the rest of the details that are OS dependent. Same is true about all other standard library functions available for I/O.

There are numerous library functions available for I/O. These can be classified into three broad categories:

- | | | |
|-----|----------------------------|--|
| (a) | Console I/O -
functions | Functions to
receive input
from
keyboard and
write output
to VDU. |
| (b) | File I/O -
functions | Functions to
perform I/O
operations on
a floppy disk
or hard disk. |

Console I/O Functions

The screen and keyboard together are called a console. Console I/O functions can be further classified into two categories — formatted and unformatted console I/O functions. The basic difference between them is that the formatted functions allow the input read from the keyboard or the output displayed on the VDU to be formatted as per our requirements. For example, if values of average marks and percentage marks are to be displayed on the screen, then the details like where this output would appear on the screen, how many spaces would be present between the two values, the number of places after the decimal points, etc. can be controlled using formatted functions. The functions available under each of these two categories are shown in Figure 5. Now let us discuss these console I/O functions in detail.

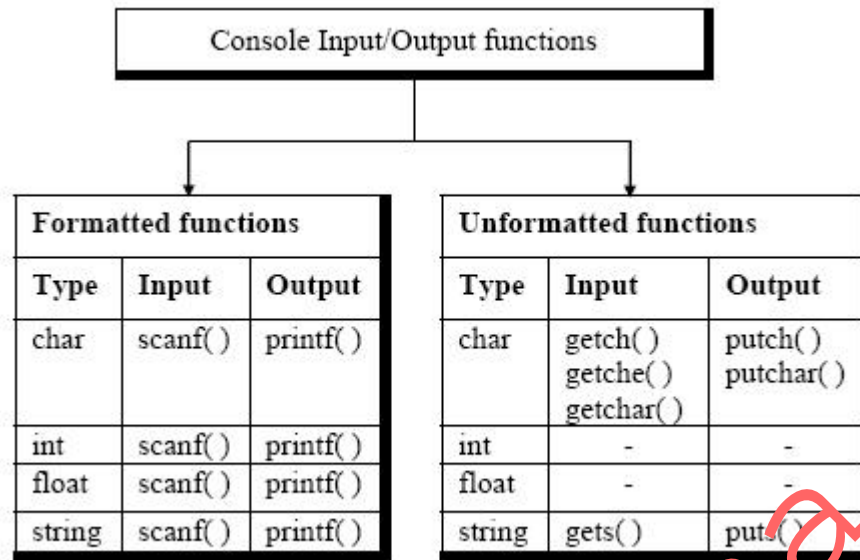


Figure 5

Formatted Console I/O Functions

As can be seen from Figure 5 the functions **printf()** and **scanf()** fall under the category of formatted console I/O functions. These functions allow us to supply the input in a fixed format and let us obtain the output in the specified form. Let us discuss these functions one by one.

We have talked a lot about **printf()**, used it regularly, but without having introduced it formally. Well, better late than never. Its general form looks like this...

printf ("format string", list of variables) ;

The format string can contain:

- a) Characters that are simply printed as they are
- b) Conversion specifications that begin with a % sign
- c) Escape sequences that begin with a \ sign

For example, look at the following program:

```
main()
{
    int avg = 346;
    float per = 69.2;
    printf ( "Average = %d\nPercentage = %f", avg, per );
}
```

The output of the program would be...

```
Average = 346
Percentage = 69.200000
```

Format Specifications

The **%d** and **%f** used in the **printf()** are called format specifiers. They tell **printf()** to print the value of **avg** as a decimal integer and the value of **per** as a float. Following is the list of format specifiers that can be used with the **printf()** function.

Data type		Format specifier
Integer	short signed	%d or %i
	short unsigned	%u
	long signed	%ld
	long unsigned	%lu
	unsigned hexadecimal	%x
	unsigned octal	%o
Real	float	%f
	double	%lf
Character	signed character	%c
	unsigned character	%c
String		%s

Escape Sequences

We saw earlier how the new line character, **\n**, when inserted in a **printf()**'s format string, takes the cursor to the beginning of the next line. The newline character is an 'escape sequence', so called because the backslash symbol (****) is considered as an 'escape' character—it causes an escape from the normal interpretation of a string, so that the next character is recognized as one having a special meaning.

Esc. Seq.	Purpose	Esc. Seq.	Purpose
\n	New line	\t	Tab
\b	Backspace	\r	Carriage return
\f	Form feed	\a	Alert
\'	Single quote	\"	Double quote
\\	Backslash		

Unformatted Console I/O Functions

There are several standard library functions available under this category—those that can deal with a single character and those that can deal with a string of characters. For openers let us look at those which handle one character at a time.

So far for input we have consistently used the **scanf()** function. However, for some situations the **scanf()** function has one glaring weakness... you need to hit the Enter key before the function can digest what you have typed. However, we often want a function that will read a single character the instant it is typed without waiting for the Enter key to be hit. **getch()** and **getche()** are two functions which serve this purpose. These functions return the character that has been most recently typed. The 'e' in **getche()** function means it echoes (displays) the character that you typed to the screen. As against this **getch()** just returns the character that you typed without echoing it on the screen. **getchar()** works similarly and echoes the character that you typed on the screen, but unfortunately requires Enter key to be typed following the character that you typed. The difference between **getchar()** and **fgetchar()** is that the former is a macro whereas the latter is a function. Here is a sample program that illustrates the use of these functions.

```
main( )
{
    char ch ;
    printf ( "\nPress any key to continue" ) ;
    getch( ) ; /* will not echo the character */
    printf ( "\nType any character" ) ;
    ch = getche( ) ; /* will echo the character typed */
    printf ( "\nType any character" ) ;
    getchar( ) ; /* will echo character, must be followed by enter key */
    printf ( "\nContinue Y/N" ) ;
    fgetchar( ) ; /* will echo character, must be followed by enter key */
}
```

And here is a sample run of this program...

```
Press any key to continue
Type any character B
Type any character W
Continue Y/N Y
```

Decision making and branching

Decisions! Decisions!

In the previous programs we have used sequence control structure in which the various steps are executed sequentially, i.e. in the same order in which they appear in the program. In fact to execute the instructions sequentially, we don't have to do anything at all. By default the instructions in a program are executed sequentially. However, in serious programming situations, seldom do we want the instructions to be executed sequentially. Many a times, we want a set of instructions to be executed in one situation, and an entirely different set of instructions to be executed in another situation. This kind of situation is dealt in C programs using a decision control instruction. As mentioned earlier, a decision control instruction can be implemented in C using:

- a) The **if** statement
- b) The **if-else** statement
- c) The conditional operators

The **if** Statement

Like most languages, C uses the keyword **if** to implement the decision control instruction. The general form of **if** statement looks like this:

```
if ( this condition is true )  
    execute this statement ;
```

The keyword **if** tells the compiler that what follows is a decision control instruction. The condition following the keyword **if** is always enclosed within a pair of parentheses. If the condition, whatever it is, is true, then the statement is executed. If the condition is not true then the statement is not executed, instead the program skips past it.

Multiple Statements within **if**

It may so happen that in a program we want more than one statement to be executed if the expression following **if** is satisfied. If such multiple statements are to be executed then they must be placed within a pair of braces as illustrated in the following example.

Example: The current year and the year in which the employee joined the organization are entered through the keyboard. If the number of years for which the employee has served the organization is greater than 3 then a bonus of Rs. 2500/- is given to the employee. If the years of service are not greater than 3, then the program should do nothing.

```
/* Calculation of bonus */
```

```
main( )
```

```

{
int bonus, cy, yoj, yr_of_ser ;
printf ( "Enter current year and year of joining " ) ;
scanf ( "%d %d", &cy, &yoy ) ;
yr_of_ser = cy - yoj ;
if ( yr_of_ser > 3 )
{
bonus = 2500 ;
printf ( "Bonus = Rs. %d", bonus ) ;
}
}

```

Observe that here the two statements to be executed on satisfaction of the condition have been enclosed within a pair of braces. If a pair of braces is not used then the C compiler assumes that the programmer wants only the immediately next statement after the **if** to be executed on satisfaction of the condition. In other words we can say that the default scope of the **if** statement is the immediately next statement after it.

The *if-else* Statement

The **if** statement by itself will execute a single statement, or a group of statements, when the expression following **if** evaluates to true. It does nothing when the expression evaluates to false. Can we execute one group of statements if the expression evaluates to true and another group of statements if the expression evaluates to false? Of course! This is what is the purpose of the **else** statement that is demonstrated in the following example:

Example: In a company an employee is paid as under:

If his basic salary is less than Rs. 1500, then HRA = 10% of basic salary and DA = 90% of basic salary. If his salary is either equal to or above Rs. 1500, then HRA = Rs. 500 and DA = 98% of basic salary. If the employee's salary is input through the keyboard write a program to find his gross salary.

/* Calculation of gross salary */

```

main( )
{
float bs, gs, da, hra ;
printf ( "Enter basic salary " ) ;
scanf ( "%f", &bs ) ;
if ( bs < 1500 )
{
hra = bs * 10 / 100 ;
da = bs * 90 / 100 ;
}
else

```



```

{
hra = 500 ;
da = bs * 98 / 100 ;
}
gs = bs + hra + da ;
printf ( "gross salary = Rs. %f", gs ) ;
}

```

Nested if-elses

It is perfectly all right if we write an entire **if-else** construct within either the body of the **if** statement or the body of an **else** statement. This is called ‘nesting’ of **ifs**. This is shown in the following program.

/* A quick demo of nested if-else */

```

main( )
{
int i ;
printf ( "Enter either 1 or 2 " ) ;
scanf ( "%d", &i ) ;
if ( i == 1 )
printf ( "You would go to heaven !" ) ;
else
{
if ( i == 2 )
printf ( "Hell was created with you in mind" ) ;
else
printf ( "How about mother earth !" ) ;
}
}

```

The else if Clause

There is one more way in which we can write program. This involves usage of **else if** blocks as shown below:

/* else if ladder demo */

```

main( )
{
int m1, m2, m3, m4, m5, per ;
per = ( m1+ m2 + m3 + m4+ m5 ) / per ;
if ( per >= 60 )
printf ( "First division" ) ;

```

```

else if ( per >= 50 )
printf ( "Second division" );
else if ( per >= 40 )
printf ( "Third division" );
else
printf ( "fail" );
}

```

The Conditional Operators (?)

The conditional operators **?** and **:** are sometimes called ternary operators since they take three arguments. In fact, they form a kind of foreshortened if-then-else. Their general form is,
 expression 1 ? expression 2 : expression 3

What this expression says is: “if **expression 1** is true (that is, if its value is non-zero), then the value returned will be **expression 2**, otherwise the value returned will be **expression 3**”.

Let us understand this with the help of a few examples:

(a) `int x, y ;`
`scanf ("%d", &x) ;`
`y = (x > 5 ? 3 : 4) ;`
 This statement will store 3 in **y** if **x** is greater than 5, otherwise it will store 4 in **y**.
 The equivalent **if** statement will be,
`if (x > 5)`
`y = 3 ;`
`else`
`y = 4 ;`

(b) `char a ;`
`int y ;`
`scanf ("%c", &a) ;`
`y = (a >= 65 && a <= 90 ? 1 : 0) ;`

Here 1 would be assigned to **y** if **a >=65 && a <=90** evaluates to true, otherwise 0 would be assigned.

The following points may be noted about the conditional operators:

(a) (b) (c) (a)

a) It's not necessary that the conditional operators should be used only in arithmetic statements. This is illustrated in the following examples:

Ex.: `int i ;`
`scanf ("%d", &i) ;`

```
( i == 1 ? printf ( "Amit" ) : printf ( "All and sundry" ) );  
Ex.: char a = 'z' ;  
printf ( "%c" , ( a >= 'a' ? a : '!' ) );
```

- b) The conditional operators can be nested as shown below.

```
int big, a, b, c ;  
big = ( a > b ? ( a > c ? 3 : 4 ) : ( b > c ? 6 : 8 ) );
```

- c) Check out the following conditional expression:

```
a > b ? g = a : g = b ;
```

This will give you an error 'Lvalue Required'. The error can be overcome by enclosing the statement in the : part within a pair of parenthesis. This is shown below:

```
a > b ? g = a : ( g = b ) ;
```

In absence of parentheses the compiler believes that `g` is being assigned to the result of the expression to the left of second `=`. Hence it reports an error.

The limitation of the conditional operators is that after the `?` or after the `:` only one C statement can occur. In practice rarely is this the requirement. Therefore, in serious C programming conditional operators aren't as frequently used as the `if-else`.

Decisions Using *switch*

The control statement that allows us to make a decision from the number of choices is called a **switch**, or more correctly a **switch case-default**, since these three keywords go together to make up the control statement. They most often appear as follows:

```
switch ( integer expression )  
{  
  case constant 1 :  
    do this ;  
  case constant 2 :  
    do this ;  
  case constant 3 :  
    do this ;  
  default :  
    do this ;  
}
```

The integer expression following the keyword **switch** is any C expression that will yield an integer value. It could be an integer constant like 1, 2 or 3, or an expression that evaluates to an

integer. The keyword **case** is followed by an integer or a character constant. Each constant in each **case** must be different from all the others. The “do this” lines in the above form of **switch** represent any valid C statement.

What happens when we run a program containing a **switch**? First, the integer expression following the keyword **switch** is evaluated. The value it gives is then matched, one by one, against the constant values that follow the **case** statements. When a match is found, the program executes the statements following that **case**, and all subsequent **case** and **default** statements as well. If no match is found with any of the **case** statements, only the statements following the **default** are executed. A few examples will show how this control structure works.

Consider the following program:

```
main( )
{
  int i = 2 ;
  switch ( i )
  {
    case 1 :
      printf ( "I am in case 1 \n" ) ;
    case 2 :
      printf ( "I am in case 2 \n" ) ;
    case 3 :
      printf ( "I am in case 3 \n" ) ;
    default :
      printf ( "I am in default \n" ) ;
  }
}
```

The output of this program would be:

```
I am in case 2
I am in case 3
I am in default
```

The output is definitely not what we expected! We didn't expect the second and third line in the above output. The program prints case 2 and 3 and the default case. Well, yes. We said the **switch** executes the case where a match is found and all the subsequent **cases** and the **default** as well.

If you want that only case 2 should get executed, it is upto you to get out of the **switch** then and there by using a **break** statement. The following example shows how this is done. Note that there is no need for a **break** statement after the **default**, since the control comes out of the **switch** anyway.

```
main( )
```

```

{
int i = 2 ;
switch ( i )
{
    case 1 :
        printf ( "I am in case 1 \n" ) ;
        break ;
    case 2 :
        printf ( "I am in case 2 \n" ) ;
        break ;
    case 3 :
        printf ( "I am in case 3 \n" ) ;
        break ;
    default :
        printf ( "I am in default \n" ) ;
}
}

```

The output of this program would be:

I am in case 2

***switch* Versus *if-else* Ladder**

There are some things that you simply cannot do with a **switch**. These are:

- A float expression cannot be tested using a **switch**
- Cases can never have variable expressions (for example it is wrong to say **case a +3 :**)
- Multiple cases cannot use same expressions. Thus the following **switch** is illegal:

The *goto* Keyword

Avoid **goto** keyword! They make a C programmer's life miserable. There is seldom a legitimate reason for using **goto**, and its use is one of the reasons that programs become unreliable, unreadable, and hard to debug. And yet many programmers find **goto** seductive.

In a difficult programming situation it seems so easy to use a **goto** to take the control where you want. However, almost always, there is a more elegant way of writing the same program using **if**, **for**, **while** and **switch**. These constructs are far more logical and easy to understand.

The big problem with **gotos** is that when we do use them we can never be sure how we got to a certain point in our code. They obscure the flow of control. So as far as possible skip them. You can always get the job done without them. Trust me, with good programming skills **goto** can always be avoided. This is the first and last time that we are going to use **goto** in this book. However, for sake of completeness of the book, the following program shows how to use **goto**.

```
main( )
{
    int goals ;
    printf ( "Enter the number of goals scored against India" ) ;
    scanf ( "%d", &goals ) ;
    if ( goals <= 5 )
        goto sos ;
    else
    {
        printf ( "About time soccer players learnt C\n" ) ;
        printf ( "and said goodbye! adieu! to soccer" ) ;
        exit( ) ; /* terminates program execution */
    }
    sos :
    printf ( "To err is human!" ) ;
}
```

And here are two sample runs of the program...

```
Enter the number of goals scored against India 5
To err is human!
Enter the number of goals scored against India 7
About time soccer players learnt C
and said goodbye! adieu! to soccer
```

A few remarks about the program would make the things clearer.

- If the condition is satisfied the **goto** statement transfers control to the label ‘sos’, causing **printf()** following **sos** to be executed.
- The label can be on a separate line or on the same line as the statement following it, as in, `sos : printf ("To err is human!") ;`
- Any number of **gotos** can take the control to the same label.
- The **exit()** function is a standard library function which terminates the execution of the program. It is necessary to use this function since we don't want the statement `printf ("To err is human!")` to get executed after execution of the **else** block.

Loops

The versatility of the computer lies in its ability to perform a set of instructions repeatedly. This involves repeating some portion of the program either a specified number of times or until a particular condition is being satisfied. This repetitive operation is done through a loop control instruction.

There are three methods by way of which we can repeat a part of a program. They are:

- (a) Using a **for** statement
- (b) Using a **while** statement
- (c) Using a **do-while** statement

The *while* loop

The most basic loop in C is the while loop. A while statement is like a repeating if statement. Like an If statement, if the test condition is true: the statements get executed. The difference is that after the statements have been executed the test condition is checked again. If it is still true the statements get executed again. This cycle repeats until the test condition evaluates to false.

Basic syntax of while loop is as follows:

```
while ( expression )  
{  
    Single statement  
    or  
    Block of statements;  
}
```

Try following example to understand **while** loop. You can put the following code into a test.c file and then compile it and then run it.

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

```

{
    int i = 10;

    while ( i > 0 )
    {
        printf("Hello %d\n", i );
        i = i -1;
    }
}

```

This will produce following output:

```

Hello 10
Hello 9
Hello 8
Hello 7
Hello 6
Hello 5
Hello 4
Hello 3
Hello 2
Hello 1

```

You can make use of **break** to come out of while loop at any time.

```

#include <stdio.h>

main()
{
    int i = 10;

    while ( i > 0 )
    {
        printf("Hello %d\n", i );
        i = i -1;
        if( i == 6 )
        {
            break;
        }
    }
}

```

This will produce following output:

Hello 10
Hello 9
Hello 8
Hello 7

for loop

for loop is similar to while, it's just written differently. for statements are often used to process lists such a range of numbers:

Basic syntax of for loop is as follows:

```
for( expression1; expression2; expression3)
{
    Single statement
    or
    Block of statements;
}
```

In the above syntax:

- expression1 - Initialises variables
- expression2 - Conditional expression, as long as this condition is true, loop will keep executing.
- expression3 - expression3 is the modifier which may be simple increment of a variable.
- Try following example to understand **for** loop. You can put the following code into a test.c file and then compile it and then run it.

```
#include <stdio.h>

main()
{
    int i;
    int j = 10;
    for( i = 0; i <= j; i ++ )
    {
        printf("Hello %d\n", i );
    }
}
```

- This will produce following output:

Hello 0
Hello 1
Hello 2
Hello 3
Hello 4
Hello 5
Hello 6
Hello 7
Hello 8
Hello 9
Hello 10

- You can make use of **break** to come out of for loop at any time.

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

```
main()
{
    int i;

    for( i = 0; i <= j; i ++ )
    {
        printf("Hello %d\n", i );
        if( i == 6 )
        {
            break;
        }
    }
}
```

- This will produce following output:

Hello 1
Hello 2
Hello 3
Hello 4
Hello 5

do...while loop

do ... while is just like a while loop except that the test condition is checked at the end of the loop rather than the start. This has the effect that the content of the loop are always executed at least once.

Basic syntax of do...while loop is as follows:

```
do
{
    Single statement
    or
    Block of statements;
}while(expression);
```

Try following example to understand **do...while** loop. You can put the following code into a test.c file and then compile it and then run it.

```
#include <stdio.h>

main()
{
    int i = 10;

    do{
        printf("Hello %d\n", i );
        i = i -1;
    }while ( i > 0 );
}
```

This will produce following output:

```
Hello 10
Hello 9
Hello 8
Hello 7
Hello 6
Hello 5
Hello 4
Hello 3
Hello 2
Hello 1
```

You can make use of **break** to come out of do...while loop at any time.

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

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

    do{
        printf("Hello %d\n", i );
        i = i -1;
        if( i == 6 )
        {
            break;
        }
    }while ( i > 0 );
}
```

This will produce following output:

```
Hello 10
Hello 9
Hello 8
Hello 7
Hello 6
```

break and continue statements

C provides two commands to control how we loop:

- **break** -- exit from loop or switch.
- **continue** -- skip 1 iteration of loop.

You already have seen example of using break statement. Here is an example showing usage of **continue** statement.

```
#include
```

```
main()
{
```

```

int i;
int j = 10;

for( i = 0; i <= j; i ++ )
{
    if( i == 5 )
    {
        continue;
    }
    printf("Hello %d\n", i );
}

```

This will produce following output:

```

Hello 0
Hello 1
Hello 2
Hello 3
Hello 4
Hello 6
Hello 7
Hello 8
Hello 9
Hello 10

```

Functions

A function is a module or block of program code which deals with a particular task. Making functions is a way of isolating one block of code from other independent blocks of code.

Functions serve two purposes.

- They allow a programmer to say: 'this piece of code does a specific job which stands by itself and should not be mixed up with anything else',
- Second they make a block of code reusable since a function can be reused in many different contexts without repeating parts of the program text.

A function can take a number of parameters, do required processing and then return a value. There may be a function which does not return any value.

You already have seen couple of built-in functions like printf(); Similar way you can define your own functions in C language.

Consider the following chunk of code

```
int total = 10;
printf("Hello World");
total = total + 1;
```

To turn it into a function you simply wrap the code in a pair of curly brackets to convert it into a single compound statement and write the name that you want to give it in front of the brackets:

```
Demo()
{
    int total = 10;
    printf("Hello World");
    total = total + 1;
}
```

curved brackets after the function's name are required. You can pass one or more parameters to a function as follows:

```
Demo( int par1, int par2)
{
    int total = 10;
    printf("Hello World");
    total = total + 1;
}
```

By default function does not return anything. But you can make a function to return any value as follows:

```
int Demo( int par1, int par2)
{
    int total = 10;
    printf("Hello World");
    total = total + 1;
    return total;
}
```

A *return* keyword is used to return a value and datatype of the returned value is specified before the name of function. In this case function returns *total* which is *int* type. If a function does not return a value then *void* keyword can be used as return value.

Once you have defined your function you can use it within a program:

```
main()
{
    Demo();
}
```

Functions and Variables:

Each function behaves the same way as C language standard function *main()*. So a function will have its own local variables defined. In the above example *total* variable is local to the function *Demo*.

A global variable can be accessed in any function in similar way it is accessed in *main()* function.

Declaration and Definition

When a function is defined at any place in the program then it is called function definition. At the time of definition of a function actual logic is implemented with-in the function.

A function declaration does not have any body and they just have their interfaces.

A function declaration is usually declared at the top of a C source file, or in a separate header file.

A function declaration is sometime called function prototype or function signature. For the above *Demo()* function which returns an integer, and takes two parameters a function declaration will be as follows:

```
int Demo(int par1, int par2);
```

Passing Parameters to a Function

There are two ways to pass parameters to a function:

- **Pass by Value:** mechanism is used when you don't want to change the value of passed parameters. When parameters are passed by value then functions in C create copies of the passed in variables and do required processing on these copied variables.
- **Pass by Reference** mechanism is used when you want a function to do the changes in passed parameters and reflect those changes back to the calling function. In this case only addresses of the variables are passed to a function so that function can work directly over the addresses.

Here are two programs to understand the difference: First example is for *Pass by value*:

```
#include <stdio.h>

/* function declaration goes here.*/
void swap( int p1, int p2 );

int main()
{
    int a = 10;
    int b = 20;

    printf("Before: Value of a = %d and value of b = %d\n", a, b );
    swap( a, b );
    printf("After: Value of a = %d and value of b = %d\n", a, b );
}

void swap( int p1, int p2 )
{
    int t;

    t = p2;
    p2 = p1;
    p1 = t;
    printf("Value of a (p1) = %d and value of b(p2) = %d\n", p1, p2 );
}
```

Here is the result produced by the above example. Here the values of a and b remain unchanged before calling *swap* function and after calling *swap* function.

```
Before: Value of a = 10 and value of b = 20
Value of a (p1) = 20 and value of b(p2) = 10
After: Value of a = 10 and value of b = 20
```

Following is the example which demonstrate the concept of pass by reference


```

#include <stdio.h>

/* function declaration goes here.*/
void swap( int *p1, int *p2 );

int main()
{
    int a = 10;
    int b = 20;

    printf("Before: Value of a = %d and value of b = %d\n", a, b );
    swap( &a, &b );
    printf("After: Value of a = %d and value of b = %d\n", a, b );
}

void swap( int *p1, int *p2 )
{
    int t;

    t = *p2;
    *p2 = *p1;
    *p1 = t;
    printf("Value of a (p1) = %d and value of b(p2) = %d\n", *p1, *p2 );
}

```

Here is the result produced by the above example. Here the values of a and b are changes after calling *swap* function.

Before: Value of a = 10 and value of b = 20
 Value of a (p1) = 20 and value of b(p2) = 10
 After: Value of a = 20 and value of b = 10

ARRAY

an array is a collection of similar elements. These similar elements could be all **ints**, or all **floats**, or all **chars**, etc. Usually, the array of characters is called a 'string', whereas an array of **ints** or **floats** is called simply an array. Remember that all elements of any given array must be of the same type. i.e. we cannot have an array of 10 numbers, of which 5 are **ints** and 5 are **floats**.

A Simple Program Using Array

Let us try to write a program to find average marks obtained by a class of 30 students in a test.

```
main( )
{ int avg, sum = 0 ;
  int i ;
  int marks[30] ;

  /* array declaration */

  for ( i = 0 ; i <= 29 ; i++ )
    { printf ( "\nEnter marks " ) ;
      scanf ( "%d", &marks[i] ) ; /* store data in array */
    }
  for ( i = 0 ; i <= 29 ; i++ )

    sum = sum + marks[i] ; /* read data from an array*/

  avg = sum / 30 ;
  printf ( "\nAverage marks = %d", avg ) ;
}
```

There is a lot of new material in this program, so let us take it apart slowly.

Array Declaration

To begin with, like other variables an array needs to be declared so that the compiler will know what kind of an array and how large an array we want. In our program we have done this with the statement:

```
int marks[30] ;
```

Here, **int** specifies the type of the variable, just as it does with ordinary variables and the word **marks** specifies the name of the variable. The **[30]** however is new. The number 30 tells how many elements of the type **int** will be in our array. This number is often called the 'dimension' of the array. The bracket ([]) tells the compiler that we are dealing with an array.

Accessing Elements of an Array

Once an array is declared, let us see how individual elements in the array can be referred. This is done with a subscript, the number in the brackets following the array name. This number specifies the element's position in the array. All the array elements are numbered, starting with 0. Thus, **marks[2]** is not the second element of the array, but the third. In our program we are using the variable **i** as a subscript to refer to various elements of the array. This variable can take different values and hence can refer to the different elements in the array in turn. This ability to use variables as subscripts is what makes arrays so useful.

Entering Data into an Array

Here is the section of code that places data into an array:

```
for ( i = 0 ; i <= 29 ; i++ )
{
    printf ( "\nEnter marks " ) ;
    scanf ( "%d", &marks[i] ) ;
}
```

The **for** loop causes the process of asking for and receiving a student's marks from the user to be repeated 30 times. The first time through the loop, **i** has a value 0, so the **scanf()** function will cause the value typed to be stored in the array element **marks[0]**, the first element of the array. This process will be repeated until **i** becomes 29. This is last time through the loop, which is a good thing, because there is no array element like

marks[30].

In **scanf()** function, we have used the “address of” operator (**&**) on the element **marks[i]** of the array, just as we have used it earlier on other variables (**&rate**, for example). In so doing, we are passing the address of this particular array element to the **scanf()** function, rather than its value; which is what **scanf()** requires.

Reading Data from an Array

The balance of the program reads the data back out of the array and uses it to calculate the average. The **for** loop is much the same, but now the body of the loop causes each student's marks to be added to a running total stored in a variable called **sum**. When all the marks have been added up, the result is divided by 30, the number of students, to get the average.

```
for ( i = 0 ; i <= 29 ; i++ )
    sum = sum + marks[i] ;
avg = sum / 30 ;
printf ( "\nAverage mark = %d", avg ) ;
```

To fix our ideas, let us revise whatever we have learnt about arrays:

- a) An array is a collection of similar elements.
- b) The first element in the array is numbered 0, so the last element is 1 less than the size of the array.
- c) An array is also known as a subscripted variable.
- d) Before using an array its type and dimension must be declared.

- e) However big an array its elements are always stored in contiguous memory locations. This is a very important point which we would discuss in more detail later on.

Array Initialization

So far we have used arrays that did not have any values in them to begin with. We managed to store values in them during program execution. Let us now see how to initialize an array while declaring it. Following are a few examples that demonstrate this.

```
int num[6] = { 2, 4, 12, 5, 45, 5 } ;  
int n[ ] = { 2, 4, 12, 5, 45, 5 } ;  
float press[ ] = { 12.3, 34.2 -23.4, -11.3 } ;
```

Note the following points carefully:

- (a) Till the array elements are not given any specific values, they are supposed to contain garbage values.
- (b) If the array is initialised where it is declared, mentioning the dimension of the array is optional as in the 2nd example above.

Array Elements in Memory

Consider the following array declaration:

```
int arr[8] ;
```

What happens in memory when we make this declaration? 16 bytes get immediately reserved in memory, 2 bytes each for the 8 integers (under Windows/Linux the array would occupy 32 bytes as each integer would occupy 4 bytes). And since the array is not being initialized, all eight values present in it would be garbage values. This so happens because the storage class of this array is assumed to be **auto**. If the storage class is declared to be **static** then all the array elements would have a default initial value as zero. Whatever be the initial values, all the array elements would always be present in contiguous memory locations. This arrangement of array elements in memory is shown in Figure.

12	34	66	-45	23	346	77	90
65508	65510	65512	65514	65516	65518	65520	65522

Bounds Checking

In C there is no check to see if the subscript used for an array exceeds the size of the array. Data entered with a subscript exceeding the array size will simply be placed in memory outside the array; probably on top of other data, or on the program itself. This will lead to unpredictable results, to say the least, and there will be no error message to warn you that you are going beyond the array size. In some cases the computer may just hang. Thus, the following program may turn out to be suicidal.

```
main( )
{
    int num[40], i ;
    for ( i = 0 ; i <= 100 ; i++ )

        num[i] = i ;

}
```

Thus, to see to it that we do not reach beyond the array size is entirely the programmer's botheration and not the compiler's.

Passing Array Elements to a Function

Array elements can be passed to a function by calling the function by value, or by reference. In the call by value we pass values of array elements to the function, whereas in the call by reference we pass addresses of array elements to the function. These two calls are illustrated below:

/* Demonstration of call by value */

```
main( )
{
    int i ;
    int marks[ ] = { 55, 65, 75, 56, 78, 78, 90 } ;

    for ( i = 0 ; i <= 6 ; i++ ) display ( marks[i] ) ;
}

display ( int m )
{
    printf ( "%d ", m ) ;
}
```

And here's the output...

55 65 75 56 78 78 90

Here, we are passing an individual array element at a time to the function **display()** and getting it printed in the function **display()**. Note that since at a time only one element is being passed, this element is collected in an ordinary integer variable **m**, in the function **display()**.

And now the call by reference.

```
/* Demonstration of call by reference */
```

```
main( )
```

```
{ int i ;  
int marks[ ] = { 55, 65, 75, 56, 78, 78, 90 } ;
```

```
for ( i = 0 ; i <= 6 ; i++ )
```

```
disp ( &marks[i] ) ;  
}
```

```
disp ( int *n )
```

```
{  
printf ( "%d ", *n ) ;  
}
```

And here's the output...

```
55 65 75 56 78 78 90
```

Pointers and Arrays

To be able to see what pointers have got to do with arrays, let us first learn some pointer arithmetic. Consider the following example:

```
main( )
```

```
{  
int i = 3, *x ;  
float j = 1.5, *y ;  
char k = 'c', *z ;  
printf ( "\nValue of i = %d", i ) ;  
printf ( "\nValue of j = %f", j ) ;  
printf ( "\nValue of k = %c", k ) ;  
x = &i ;  
y = &j ;
```

```

z = &k ;
printf ( "\nOriginal address in x = %u", x ) ;
printf ( "\nOriginal address in y = %u", y ) ;
printf ( "\nOriginal address in z = %u", z ) ;
x++ ;
y++ ;
z++ ;
printf ( "\nNew address in x = %u", x ) ;
printf ( "\nNew address in y = %u", y ) ;
printf ( "\nNew address in z = %u", z ) ;
}

```

Here is the output of the program.

```

Value of i = 3
Value of j = 1.500000
Value of k = c
Original address in x = 65524
Original address in y = 65520
Original address in z = 65519
New address in x = 65526
New address in y = 65524
New address in z = 65520

```

Passing an Entire Array to a Function

In the previous section we saw two programs—one in which we passed individual elements of an array to a function, and another in which we passed addresses of individual elements to a function. Let us now see how to pass an entire array to a function rather than its individual elements. Consider the following example:

```

/* Demonstration of passing an entire array to a function */

```

```

main( )
{
    int num[ ] = { 24, 31, 12, 44, 56, 17 } ;
    display ( &num, 6 ) ;
}

```

```

display ( int *j, int n )

```

```

{
    int i ;

    for ( i = 0 ; i <= n - 1 ; i++ )
    {

```

```
printf ( "\nelement = %d", *j ) ;
j++ ; /* increment pointer to point to next element */
}
}
```

Here, the **display()** function is used to print out the array elements. Note that the address of the zeroth element is being passed to the **display()** function. The **for** loop is same as the one used in the earlier program to access the array elements using pointers. Thus, just passing the address of the zeroth element of the array to a function is as good as passing the entire array to the function. It is also necessary to pass the total number of elements in the array, otherwise the **display()** function would not know when to terminate the **for** loop. Note that the address of the zeroth element (many a times called the base address) can also be passed by just passing the name of the array. Thus, the following two function calls are same:

```
display ( &num[0], 6 ) ;
```

```
display ( num, 6 ) ;
```

Two Dimensional Arrays

So far we have explored arrays with only one dimension. It is also possible for arrays to have two or more dimensions. The two-dimensional array is also called a matrix.

Here is a sample program that stores roll number and marks obtained by a student side by side in a matrix.

```
main( )
{
    int stud[4][2] ;
    int i, j ;

    for ( i = 0 ; i <= 3 ; i++ )
    {
```



```
printf ( "\n Enter roll no. and marks" ) ;
scanf ( "%d %d", &stud[i][0], &stud[i][1] ) ;
}
```

```
for ( i = 0 ; i <= 3 ; i++ )
printf ( "\n%d %d", stud[i][0], stud[i][1] ) ;
}
```

There are two parts to the program—in the first part through a **for** loop we read in the values of roll no. and marks, whereas, in second part through another **for** loop we print out these values. Look at the **scanf()** statement used in the first **for** loop:

```
scanf ( "%d %d", &stud[i][0], &stud[i][1] ) ;
```

In **stud[i][0]** and **stud[i][1]** the first subscript of the variable **stud**, is row number which changes for every student. The second subscript tells which of the two columns are we talking about—the zeroth column which contains the roll no. or the first column which contains the marks.

Remember the counting of rows and columns begin with zero. The complete array arrangement is shown below.

	col. no. 0	col. no. 1
row no. 0	1234	56
row no. 1	1212	33
row no. 2	1434	80
row no. 3	1312	78

so on. The above arrangement highlights the fact that a two- dimensional array is nothing but a collection of a number of one- dimensional arrays placed one below the other.

In our sample program the array elements have been stored rowwise and accessed rowwise. However, you can access the array elements columnwise as well. Traditionally, the array elements are being stored and accessed rowwise; therefore we would also stick to the same strategy.

Initialising a 2-Dimensional Array

How do we initialize a two-dimensional array? As simple as this...

```
int stud[4][2] = {
    { 1234, 56 },
    { 1212, 33 },
```

```
    { 1434, 80 },  
    { 1312, 78 }  
};
```

or even this would work...

```
int stud[4][2] = { 1234, 56, 1212, 33, 1434, 80, 1312, 78 } ;
```

of course with a corresponding loss in readability.

It is important to remember that while initializing a 2-D array it is necessary to mention the second (column) dimension, whereas the first dimension (row) is optional.

Thus the declarations,

```
int arr[2][3] = { 12, 34, 23, 45, 56, 45 } ;
```

```
int arr[ ][3] = { 12, 34, 23, 45, 56, 45 } ;
```

are perfectly acceptable,

whereas,

```
int arr[2][ ] = { 12, 34, 23, 45, 56, 45 } ;
```

```
int arr[ ][ ] = { 12, 34, 23, 45, 56, 45 } ;
```

would never work.

Memory Map of a 2 Dimensional Array

Let us reiterate the arrangement of array elements in a two-dimensional array of students, which contains roll nos. in one column and the marks in the other.

The array arrangement shown in Figure is only conceptually true. This is because memory doesn't contain rows and columns. In memory whether it is a one-dimensional or a two-dimensional array the array elements are stored in one continuous chain. The arrangement of array elements of a two-dimensional array in memory is shown below:

s[0][0]	s[0][1]	s[1][0]	s[1][1]	s[2][0]	s[2][1]	s[3][0]	s[3][1]
1234	56	1212	33	1434	80	1312	78
65508	65510	65512	65514	65516	65518	65520	65522

We can easily refer to the marks obtained by the third student using the subscript notation as shown below:

```
printf ( "Marks of third student = %d", stud[2][1] );
```

Can we not refer the same element using pointer notation, the way we did in one-dimensional arrays? Answer is yes. Only the procedure is slightly difficult to understand. So, read on...

Pointers and 2-Dimensional Arrays

The C language embodies an unusual but powerful capability—it can treat parts of arrays as arrays. More specifically, each row of a two-dimensional array can be thought of as a one-dimensional array. This is a very important fact if we wish to access array elements of a two-dimensional array using pointers.

Thus, the declaration,

```
int s[5][2] ;
```

can be thought of as setting up an array of 5 elements, each of which is a one-dimensional array containing 2 integers. We refer to an element of a one-dimensional array using a single subscript. Similarly, if we can imagine **s** to be a one-dimensional array then we can refer to its zeroth element as **s[0]**, the next element as **s[1]** and so on. More specifically, **s[0]** gives the address of the zeroth one-dimensional array, **s[1]** gives the address of the first one-dimensional array and so on. This fact can be demonstrated by the following program.

```
/* Demo: 2-D array is an array of arrays */
```

```
main()
{
    int s[4][2] = {
        { 1234, 56 },
        { 1212, 33 },
        { 1434, 80 },
        { 1312, 78 }
```

```

        };
int i;

for ( i = 0 ; i <= 3 ; i++ )
printf ( "\nAddress of %d th 1-D array = %u", i, s[i] );

}

```

And here is the output...

```

Address of 0 th 1-D array = 65508
Address of 1 th 1-D array = 65512
Address of 2 th 1-D array = 65516
Address of 3 th 1-D array = 65520

```

Let's figure out how the program works. The compiler knows that `s` is an array containing 4 one-dimensional arrays, each containing 2 integers. Each one-dimensional array occupies 4 bytes (two bytes for each integer). These one-dimensional arrays are placed linearly (zeroth 1-D array followed by first 1-D array, etc.).

Pointer to an Array

If we can have a pointer to an integer, a pointer to a float, a pointer to a char, then can we not have a pointer to an array? We certainly can. The following program shows how to build and use it.

```

/* Usage of pointer to an array */

```

```

main()
{
int s[5][2] = {
    { 1234, 56 },
    { 1212, 33 },
    { 1434, 80 },
    { 1312, 78 }
};
int (*p)[2];
int i, j, *pint;

for ( i = 0 ; i <= 3 ; i++ )
{
    p = &s[i];

```

```

    pint = p ;
    printf ( "\n" );
    for ( j = 0 ; j <= 1 ; j++ )
        printf ( "%d ", *( pint + j ) );
    }
}

```

And here is the output...

1234 56

1212 33

1434 80

1312 78

Here **p** is a pointer to an array of two integers. Note that the parentheses in the declaration of **p** are necessary. Absence of them would make **p** an array of 2 integer pointers. Array of pointers is covered in a later section in this chapter. In the outer **for** loop each time we store the address of a new one-dimensional array. Thus first time through this loop **p** would contain the address of the zeroth 1-D array. This address is then assigned to an integer pointer **pint**. Lastly, in the inner **for** loop using the pointer **pint** we have printed the individual elements of the 1-D array to which **p** is pointing.

Array of Pointers

The way there can be an array of **ints** or an array of **floats**, similarly there can be an array of pointers. Since a pointer variable always contains an address, an array of pointers would be nothing but a collection of addresses. The addresses present in the array of pointers can be addresses of isolated variables or addresses of array elements or any other addresses. All rules that apply to an ordinary array apply to the array of pointers as well. I think a program would clarify the concept.

```

main( )
{
    int *arr[4] ; /* array of integer pointers */
    int i = 31, j = 5, k = 19, l = 71, m ;
    arr[0] = &i ;
    arr[1] = &j ;
    arr[2] = &k ;
}

```

```

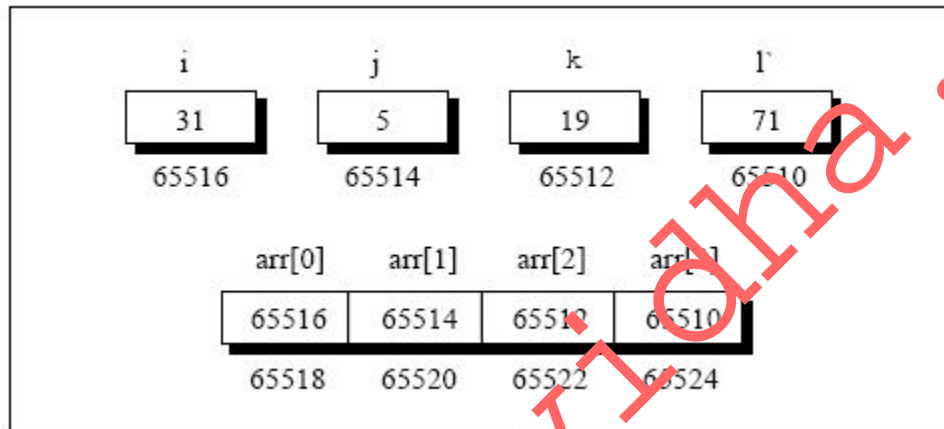
arr[3] = &l ;
for ( m = 0 ; m <= 3 ; m++ )

printf ( "%d ", * ( arr[m] ) ) ;

}

```

Figure shows the contents and the arrangement of the array of pointers in memory. As you can observe, **arr** contains addresses of isolated **int** variables **i**, **j**, **k** and **l**. The **for** loop in the program picks up the addresses present in **arr** and prints the values present at these addresses.



STRING

What are Strings

The way a group of integers can be stored in an integer array, similarly a group of characters can be stored in a character array. Character arrays are many a time also called strings. Many languages internally treat strings as character arrays, but somehow conceal this fact from the programmer. Character arrays or strings are used by programming languages to manipulate text such as words and sentences.

A string constant is a one-dimensional array of characters terminated by a null (`'\0'`). For example,

```
char name[] = { 'H', 'A', 'E', 'S', 'L', 'E', 'R', '\0' } ;
```

Each character in the array occupies one byte of memory and the last character is always `'\0'`. What character is this? It looks like two characters, but it is actually only one character, with the `\` indicating that what follows it is something special. `'\0'` is called null character. Note that `'\0'` and `'0'` are not same. ASCII value of `'\0'` is 0, whereas ASCII value of `'0'` is 48. Figure shows

the way a character array is stored in memory. Note that the elements of the character array are stored in contiguous memory locations.

The terminating null ('`\0`') is important, because it is the only way the functions that work with a string can know where the string ends. In fact, a string not terminated by a '`\0`' is not really a string, but merely a collection of characters.

H	A	E	S	L	E	R	\0
65518	65519	65520	65521	65522	65523	65524	65525

C concedes the fact that you would use strings very often and hence provides a shortcut for initializing strings. For example, the string used above can also be initialized as

```
char name[ ] = "HAESLER" ;
```

Note that, in this declaration '`\0`' is not necessary. C inserts the null character automatically.

Pointers and Strings

Suppose we wish to store "Hello". We may either store it in a string or we may ask the C compiler to store it at some location in memory and assign the address of the string in a **char** pointer. This is shown below:

```
char str[ ] = "Hello" ;  
char *p = "Hello" ;
```

There is a subtle difference in usage of these two forms. For example, we cannot assign a string to another, whereas, we can assign a **char** pointer to another **char** pointer. This is shown in the following program.

```
main( )  
{  
  char str1[ ] = "Hello" ;  
  char str2[10] ;  
  char *s = "Good Morning" ;  
  char *q ;  
  str2 = str1 ; /* error */  
  q = s ; /* works */  
}
```

Also, once a string has been defined it cannot be initialized to another set of characters. Unlike strings, such an operation is perfectly valid with **char** pointers.

```
main( )
{
char str1[ ] = "Hello" ;
char *p = "Hello" ;
str1 = "Bye" ; /* error */
p = "Bye" ; /* works */
}
```

Standard Library String Functions

With every C compiler a large set of useful string handling library functions are provided. Figure lists the more commonly used functions along with their purpose.

Function	Use
strlen	Finds length of a string
strlwr	Converts a string to lowercase
strupr	Converts a string to uppercase
strcat	Appends one string at the end of another
strncat	Appends first n characters of a string at the end of another
strcpy	Copies a string into another
strncpy	Copies first n characters of one string into another
strcmp	Compares two strings
strncmp	Compares first n characters of two strings
strcmpi	Compares two strings without regard to case ("i" denotes that this function ignores case)
stricmp	Compares two strings without regard to case (identical to strcmpi)
strnicmp	Compares first n characters of two strings without regard to case
strdup	Duplicates a string
strchr	Finds first occurrence of a given character in a string
strrchr	Finds last occurrence of a given character in a string
strstr	Finds first occurrence of a given string in another string
strset	Sets all characters of string to a given character

strnset	Sets first n characters of a string to a given character
strrev	Reverses string

Out of the above list we shall discuss the functions **strlen()**, **strcpy()**, **strcat()** and **strcmp()**, since these are the most commonly used functions. This will also illustrate how the library functions in general handle strings. Let us study these functions one by one.

strlen()

This function counts the number of characters present in a string. Its usage is illustrated in the following program.

```
main( )
{
char arr[ ] = "Bamboozled" ;
int len1, len2 ;
len1 = strlen ( arr ) ;
len2 = strlen ( "Humpty Dumpty" ) ;
printf ( "\nstring = %s length = %d", arr, len1 ) ;
printf ( "\nstring = %s length = %d", "Humpty Dumpty", len2 ) ;
}
```

The output would be...

```
string = Bamboozled length = 10
string = Humpty Dumpty length = 13
```

Note that in the first call to the function **strlen()**, we are passing the base address of the string, and the function in turn returns the length of the string. While calculating the length it doesn't count '\0'. Even in the second call,

```
len2 = strlen ( "Humpty Dumpty" ) ;
```

what gets passed to **strlen()** is the address of the string and not the string itself.

strcpy()

This function copies the contents of one string into another. The base addresses of the source and target strings should be supplied to this function. Here is an example of **strcpy()** in action...

```

main( )
{
char source[ ] = "Sayonara" ;
char target[20] ;
strcpy ( target, source ) ;
printf ( "\nsource string = %s", source ) ;
printf ( "\ntarget string = %s", target ) ;
}

```

And here is the output...

```

source string = Sayonara
target string = Sayonara

```

On supplying the base addresses, **strcpy()** goes on copying the characters in source string into the target string till it doesn't encounter the end of source string ('\0'). It is our responsibility to see to it that the target string's dimension is big enough to hold the string being copied into it. Thus, a string gets copied into another, piece-meal, character by character.

strcat()

This function concatenates the source string at the end of the target string. For example, "Bombay" and "Nagpur" on concatenation would result into a string "BombayNagpur". Here is an example of **strcat()** at work.

```

main( )
{
char source[ ] = "Folks!" ;
char target[30] = "Hello" ;
strcat ( target, source ) ;
printf ( "\nsource string = %s", source ) ;
printf ( "\ntarget string = %s", target ) ;
}

```

And here is the output...

```

source string = Folks!
target string = HelloFolks!

```

strcmp()

This is a function which compares two strings to find out whether they are same or different. The two strings are compared character by character until there is a mismatch or end of one of the strings is reached, whichever occurs first. If the two strings are identical, **strcmp()** returns a value zero. If they're not, it returns the numeric difference between the ASCII values of the first non-matching pairs of characters. Here is a program which puts **strcmp()** in action.

```
main( )
{
char string1[ ] = "Jerry" ;
char string2[ ] = "Ferry" ;
int i, j, k ;
i = strcmp ( string1, "Jerry" ) ;
j = strcmp ( string1, string2 ) ;
k = strcmp ( string1, "Jerry boy" ) ;
printf ( "\n%d %d %d", i, j, k ) ;
}
```

And here is the output...

0 4 -32

In the first call to **strcmp()**, the two strings are identical—"Jerry" and "Jerry"—and the value returned by **strcmp()** is zero. In the second call, the first character of "Jerry" doesn't match with the first character of "Ferry" and the result is 4, which is the numeric.

Structure & Union

Why Use Structures

We have seen earlier how ordinary variables can hold one piece of information and how arrays can hold a number of pieces of information of the same data type. These two data types can

handle a great variety of situations. But quite often we deal with entities that are collection of dissimilar data types.

For example, suppose you want to store data about a book. You might want to store its name (a string), its price (a float) and number of pages in it (an int). If data about say 3 such books is to be stored, then we can follow two approaches:

- a) Construct individual arrays, one for storing names, another for storing prices and still another for storing number of pages.
- b) Use a structure variable.

Let us examine these two approaches one by one. For the sake of programming convenience assume that the names of books would be single character long. Let us begin with a program that uses arrays.

```
main( )
{
char name[3] ;
float price[3] ;
int pages[3], i ;
printf ( "\nEnter names, prices and no. of pages of 3 books\n" ) ;
for ( i = 0 ; i <= 2 ; i++ )
scanf ( "%c %f %d", &name[i], &price[i], &pages[i] ) ;
printf ( "\nAnd this is what you entered\n" ) ;
for ( i = 0 ; i <= 2 ; i++ )
printf ( "%c %f %d\n", name[i], price[i], pages[i] ) ;
}
```

And here is the sample run...

Enter names, prices and no. of pages of 3 books

A 100.00 354

C 256.50 682

F 233.70 512

And this is what you entered

A 100.000000 354

C 256.500000 682

F 233.700000 512

This approach no doubt allows you to store names, prices and number of pages. But as you must have realized, it is an unwieldy approach that obscures the fact that you are dealing with a group of characteristics related to a single entity—the book.

A **structure** contains a number of data types grouped together. These data types may or may not be of the same type. The following example illustrates the use of this data type.

```
main( )
{
    struct book
    {
        char name ;
        float price ;
        int pages ;
    };

    struct book b1, b2, b3 ;

    printf ( "\nEnter names, prices & no. of pages of 3 books\n" );
    scanf ( "%c %f %d", &b1.name, &b1.price, &b1.pages );
    scanf ( "%c %f %d", &b2.name, &b2.price, &b2.pages );
    scanf ( "%c %f %d", &b3.name, &b3.price, &b3.pages );
    printf ( "\nAnd this is what you entered" );
    printf ( "\n%c %f %d", b1.name, b1.price, b1.pages );
    printf ( "\n%c %f %d", b2.name, b2.price, b2.pages );
    printf ( "\n%c %f %d", b3.name, b3.price, b3.pages );
}
```

And here is the output...

Enter names, prices and no. of pages of 3 books

A 100.00 354
C 256.50 682
F 233.70 512

And this is what you entered

A 100.000000 354
C 256.500000 682
F 233.700000 512

This program demonstrates two fundamental aspects of structures:

- (a) declaration of a structure
- (b) accessing of structure elements

Let us now look at these concepts one by one.

Declaring a Structure

In our example program, the following statement declares the structure type:

```
struct book
{
    char name ;
    float price ;
    int pages ;
};
```

This statement defines a new data type called **struct book**. Each variable of this data type will consist of a character variable called **name**, a float variable called **price** and an integer variable called **pages**. The general form of a structure declaration statement is given below:

```
struct <structure name>

{
    structure element 1 ;
    structure element 2 ;
    structure element 3 ;
    .....
    .....
};
```

Once the new structure data type has been defined one or more variables can be declared to be of that type. For example the variables **b1**, **b2**, **b3** can be declared to be of the type **struct book**, as,

```
struct book b1, b2, b3 ;
```

This statement sets aside space in memory. It makes available space to hold all the elements in the structure—in this case, 7 bytes—one for **name**, four for **price** and two for **pages**. These bytes are always in adjacent memory locations.

If we so desire, we can combine the declaration of the structure type and the structure variables in one statement.

For example,

```
struct book
{
    char name ;
    float price ;
    int pages ;
};
```

```
struct book b1, b2, b3 ;
```

is same as...

```
struct book
{
    char name ;
    float price ;
    int pages ;
} b1, b2, b3 ;
```

or even...

```
struct
{
    char name ;
    float price ;
    int pages ;

} b1, b2, b3 ;
```

Like primary variables and arrays, structure variables can also be initialized where they are declared. The format used is quite similar to that used to initiate arrays.

```
struct book
{
    char name[10] ;
    float price ;
    int pages ;
} ;

struct book b1 = { "Basic", 130.00, 550 } ;
struct book b2 = { "Physics", 150.80, 800 } ;
```

Note the following points while declaring a structure type:

- a) The closing brace in the structure type declaration must be followed by a semicolon.
- b) It is important to understand that a structure type declaration does not tell the compiler to reserve any space in memory. All a structure declaration does is, it defines the 'form' of the structure.

- c) Usually structure type declaration appears at the top of the source code file, before any variables or functions are defined. In very large programs they are usually put in a separate header file, and the file is included (using the preprocessor directive `#include`) in whichever program we want to use this structure type.

Accessing Structure Elements

Having declared the structure type and the structure variables, let us see how the elements of the structure can be accessed.

In arrays we can access individual elements of an array using a subscript. Structures use a different scheme. They use a dot (.) operator. So to refer to **pages** of the structure defined in our sample program we have to use,

`b1.pages`

Similarly, to refer to **price** we would use,

`b1.price`

Note that before the dot there must always be a structure variable and after the dot there must always be a structure element.

How Structure Elements are Stored

Whatever be the elements of a structure, they are always stored in contiguous memory locations. The following program would illustrate this:

```
/* Memory map of structure elements */
```

```
main()  
{  
    struct book  
    {  
        char name ;  
        float price ;  
        int pages ;  
    }
```



```

    };

    struct book b1 = { 'B', 130.00, 550 };

    printf ( "\nAddress of name = %u", &b1.name );

    printf ( "\nAddress of price = %u", &b1.price );

    printf ( "\nAddress of pages = %u", &b1.pages );
}

```

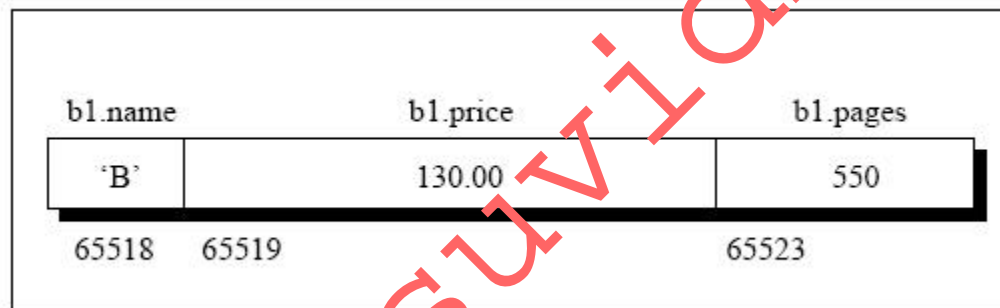
Here is the output of the program...

```

Address of name = 65518
Address of price = 65519
Address of pages = 65523

```

Actually the structure elements are stored in memory as shown in the Figure.



Array of Structures

Our sample program showing usage of structure is rather simple minded. All it does is, it receives values into various structure elements and output these values. But that's all we intended to do any way... show how structure types are created, how structure variables are declared and how individual elements of a structure variable are referenced.

In our sample program, to store data of 100 books we would be required to use 100 different structure variables from **b1** to **b100**, which is definitely not very convenient. A better approach would be to use an array of structures. Following program shows how to use an array of structures.

```

/* Usage of an array of structures */

```

```

main( )

```

```

{
    struct book
    {
        char name ;
        float price ;
        int pages ;
    } ;

    struct book b[100] ;

    int i ;

    for ( i = 0 ; i <= 99 ; i++ )
    {
        printf ( "\nEnter name, price and pages " ) ;
        scanf ( "%c %f %d", &b[i].name, &b[i].price, &b[i].pages ) ;
    }

    for ( i = 0 ; i <= 99 ; i++ )
        printf ( "\n%c %f %d", b[i].name, b[i].price, b[i].pages ) ;
    }

    linkfloat( )
    {
        float a = 0, *b ;
        b = &a ; /* cause emulator to be linked */
        a = *b ; /* suppress the warning variable not used */
    }
}

```

STRUCTURES WITHIN STRUCTURES

Structure within a structure means nesting of structures. Let us consider the following structure defined to store information about the salary of employees.

```

struct salary {
    char name[20];
    char department[10];
    int basic_pay;
    int dearness_allowance;
}

```

```

int city_allowance;
}
employee;

```

This structure defines name, department, basic pay and 3 kinds of allowance. we can group all the items related to allowance together and declare them under a substructure are shown below:

```

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

```

The salary structure contains a member named allowance which itself is a structure with 3 members. The members contained in the inner, structure namely dearness, hous_rent, and city can be referred to as :

```

employee.allowance.dearness
employee.allowance.hous_rent
employee.allowance.city

```

An inner-most member in a nested structure can be accessed by chaining all the concerned. Structure variables (from outer most to inner-most) with the member using dot operator. The following being invalid.

```

employee.allowance (actual member is missing)
employee.hous_rent (inner structure variable is missing)

```

Passing a Structure as a whole to a Function

Structures are passed to functions by way of their pointers. Thus, the changes made to the structure members inside the function will be reflected even outside the function.

```

#include <stdio.h>
typedef struct
{
char *name;
int acc_no;
char acc_types;

```

```

float balance;
} account;

main()
{
void change(account *pt);
static account person = {"chetan", 4323, 'R', 12.45};
printf("%s %d %c %.2f \n", person.
name, person. acc_type, person. acc_type, person.balance);
change(&person);
printf("%s %d %c %.2f \n", person. name,
person. acc_type, person.
acc-type, person. balance);
getch();
}

void change(account *pt)
{
pt -> name = "Rohit R";
pt -> acc_no = 1111;
pt -> acc_type = 'c';
pt -> balance = 44.12;
return;
}

```

Output

```

chetan 4323 R 12.45
Rohit R 1111 c 44.12

```

Comparing Two Structures

You can use `memcmp()`. See `man memcmp` for more details. If your structure has pointers, then move them to the end and do not compare memory area for the pointer. You have to compare pointers explicitly.

```

#include
#include
struct S{
int i;
char sz[4];
char *ptrC;
}s1, s2, s3;

```

```

void compare(void *, void *, int);
#define POPULATE(S,li,lsz,lptr) { \
S.i=li; \
memcpy(S.sz,lsz,4); \
S.sz[3]=0; \
S.ptrC = lptr; \
}
char * one = "one";
char * two = "two";
int main(int argc, char* argv[])
{
POPULATE(s1,12,"HI",one);
POPULATE(s2,12,"HI",one);
POPULATE(s3,12,"HI",two);
compare( &s1, &s2 , sizeof(s1) );
compare( &s1, &s3 , sizeof(s1) );
compare( &s1, &s3 , 8 );
return 0;
}
void compare(void *v1, void* v2, int size)
{
if( memcmp( v1, v2, size) == 0 )
printf("Memory area is same type\n");
else
printf("Memory area is different type\n");
return ;
}

```

The memcmp method needs a bit of extra caution because of memory alignment considerations.

For example:

```

struct A
{
    int n;
    char c;
    int m;
};

```

The memory layout of the struct A will be, on a 32 bit system with 4 byte alignment:

```

4 bytes for n;
1 byte for c;
3 unused bytes (garbage);
4 bytes for m;

```

If you use memcmp, it can be that the 4 + 1 + 4 used bytes are identical but that there is a difference between the 3 garbage bytes. I don't think in this case you want your comparison function to return false.

A workaround can be something like:

```
bool areEqual (struct A a1, struct A a2)
{
    struct A t1, t2;
    memset (&t1, 0, sizeof(A));
    memset (&t2, 0, sizeof(A));
    t1 = a1;
    t2 = a2;
    return !memcmp (&t1, &t2, sizeof(A));
}
```

If you have pointers in your structures, I suggest that you use a different structure for your “raw” data and add the pointers later:

```
struct A
{
    int n;
    char c;
    int m;
};

struct APTR
{
    A a;
    char* pt;
};

bool areEqualPtr (struct APTR a1, struct APTR a2)
{
    if (!areEqual (a1.a, a2.a))
        return false;
    return !strcmp (a1.pt, a2.pt);
}
```

Uses of Structures

Where are structures useful? The immediate application that comes to the mind is Database Management. That is, to maintain data about employees in an organization, books in a library, items in a store, financial accounting transactions in a company etc. But mind you, use of structures stretches much beyond database management. They can be used for a variety of purposes like:

- a. Changing the size of the cursor
- b. Clearing the contents of the screen
- c. Placing the cursor at an appropriate position on screen
- d. Drawing any graphics shape on the screen
- e. Receiving a key from the keyboard
- f. Checking the memory size of the computer
- g. Finding out the list of equipment attached to the computer
- h. Formatting a floppy
- i. Hiding a file from the directory
- j. Displaying the directory of a disk
- k. Sending the output to printer
- l. Interacting with the mouse

UNIONS

Unions, like structures contain members, whose individual data types may vary. There is major distinction between them in terms of storage. In structures each member has its own storage location, whereas all the members of a union use the same location. Like structures, a union can be declared using the keyword union as follows:

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

This declares a variable code of type union them. The union contains them members, each with a different data type. However, we can use only one of them at a time. This is due to the fact that only one location is allocated for a union variable, irrespective of its size. The compiler allocates a piece of storage that is large enough to hold the largest variable type in the union. In the declaration above, the member x requires 4 bytes which is the largest among the members. The above figure shown how all the three variables share the same address, this assumes that a float variable requires 4 bytes of storage. To access a union member, we can use the same syntax that we as for structure members, that is,

```
code. m
code. x
code. c are all valid
```

Member variables, we should make sure that we can accessing the member whose value is currently storage. For example

```
code. m = 565;
code. x = 783.65;
printf("%d", code. m); would produce erroneous output.
```

```
# include <stdio.h>
main( ){
union
{
int one;
char two;
} val;
val. one = 300;
printf("val. one = %d \n", val. one);
printf("val. two = %d \n", val. two);
}
```

The format of union is similar to structure, with the only difference in the keyword used. The above example, we have 2 members int one and char two we have then initialised the member one to 300. Here we have initialised only one member of the union. Using two printf statements, then we are displaying the individual members of the union val as:

```
val. one = 300
val. two = 44
```

As we have not initialized the char variable two, the second printf statement will give a random value of 44.

The general formats of a union thus, can be shown as.

```
union tag {
member 1;
```



```

member 2;
- - -
- - -
member
m;
};

```

The general format for defining individual union variables:

Storage-class Union tag variable 1, variable 2,....., variable n;

Storage-class and tag are optional variable 1, variable 2 etc, are union variable of type tag

Declaring union and defining variables can be done at the same time as shown below.

```

Storage-class union tag {
member 1;
member 2;
- - -
- - -
- - -member
m;
}
variable 1, variable 2, - - - , variable n;

```

Storage Classes

A storage class defines the scope (visibility) and life time of variables and/or functions within a C Program.

There are following storage classes which can be used in a C Program

- auto
- register
- static
- extern

auto - Storage Class

auto is the default storage class for all local variables.

```

{
int Count;
auto int Month;
}

```

The example above defines two variables with the same storage class. auto can only be used within functions, i.e. local variables.

register - Storage Class

register is used to define local variables that should be stored in a register instead of RAM. This means that the variable has a maximum size equal to the register size (usually one word) and can't have the unary '&' operator applied to it (as it does not have a memory location).

```

{
    register int Miles;
}

```

Register should only be used for variables that require quick access - such as counters. It should also be noted that defining 'register' goes not mean that the variable will be stored in a register. It means that it MIGHT be stored in a register - depending on hardware and implementation restrictions.

static - Storage Class

static is the default storage class for global variables. The two variables below (count and road) both have a static storage class.

```

static int Count;
int Road;

{
    printf("%d\n", Road);
}

```

static variables can be 'seen' within all functions in this source file. At link time, the static variables defined here will not be seen by the object modules that are brought in.

static can also be defined within a function. If this is done the variable is initialised at run time but is not reinitialized when the function is called. This inside a function static variable retains its value during various calls.

```

void func(void);

static count=10; /* Global variable - static is the default */

main()
{
    while (count-->0)
    {
        func();
    }
}

void func( void )
{
    static i = 5;
    i++;
    printf("i is %d and count is %d\n", i, count);
}

```

This will produce following result

```

i is 6 and count is 9
i is 7 and count is 8
i is 8 and count is 7
i is 9 and count is 6
i is 10 and count is 5
i is 11 and count is 4
i is 12 and count is 3
i is 13 and count is 2
i is 14 and count is 1
i is 15 and count is 0

```

NOTE : Here keyword *void* means function does not return anything and it does not take any parameter. You can memorise void as nothing. static variables are initialized to 0 automatically.

Definition vs Declaration : Before proceeding, let us understand the difference between *definition* and *declaration* of a variable or function. Definition means where a variable or function is defined in reality and actual memory is allocated for variable or function. Declaration means just giving a reference of a variable and function. Through declaration we assure to the compiler that this variable or function has been defined somewhere else in the program and will be provided at the time of linking. In the above examples *char *func(void)* has been put at the top which is a declaration of this function whereas this function has been defined below to *main()* function.

There is one more very important use for 'static'. Consider this bit of code.

```
char *func(void);

main()
{
    char *Text1;
    Text1 = func();
}

char *func(void)
{
    char Text2[10]="martin";
    return(Text2);
}
```

Now, 'func' returns a pointer to the memory location where 'text2' starts. But text2 has a storage class of 'auto' and will disappear when we exit the function and could be overwritten by something else. The answer is to specify

```
static char Text[10]="martin";
```

The storage assigned to 'text2' will remain reserved for the duration of the program.

extern - Storage Class

extern is used to give a reference of a global variable that is visible to ALL the program files. When you use 'extern' the variable cannot be initialized as all it does is point the variable name at a storage location that has been previously defined.

When you have multiple files and you define a global variable or function which will be used in other files also, then *extern* will be used in another file to give reference of defined variable or function. Just for understanding *extern* is used to declare a global variable or function in another files.

File 1: main.c

```
int count=5;

main()
{
    write_extern();
}
```

File 2: write.c

```
void write_extern(void);
```

```
extern int count;
```

```
void write_extern(void)
{
    printf("count is %i\n", count);
}
```

Here *extern* keyword is being used to declare count in another file.

Now compile these two files as follows

```
gcc main.c write.c -o write
```

This will produce *write* program which can be executed to produce result.

Count in 'main.c' will have a value of 5. If main.c changes the value of count - write.c will see the new value.

SECTION-D

POINTERS in C

Introduction

A variable in a program is something with a name, the value of which can vary. The way the compiler and linker handles this is that it assigns a specific block of memory within the computer to hold the value of that variable. The size of that block depends on the range over which the variable is allowed to vary. For example, on 32 bit PC's the size of an integer variable is 4 bytes. On older 16 bit PCs integers were 2 bytes. In C the size of a variable type such as an integer need not be the same on all types of machines. Further more there is more than one type of integer variable in C. We have integers, long integers and short integers which you can read up on in any basic text on C. This document assumes the use of a 32 bit system with 4 byte integers.

If you want to know the size of the various types of integers on your system, running the following code will give you that information.

```
#include <stdio.h>
int main()
{
    printf("size of a short is %d", sizeof(short));
    printf("size of a int is %d", sizeof(int));
    printf("size of a long is %d", sizeof(long));
}
```

When we declare a variable we inform the compiler of two things, the name of the variable and the type of the variable. For example, we declare a variable of type integer with the name **k** by writing:

```
int k;
```

On seeing the "int" part of this statement the compiler sets aside 4 bytes of memory (on a PC) to hold the value of the integer. It also sets up a symbol table. In that table it adds the symbol **k** and the relative address in memory where those 4 bytes were set aside.

Thus, later if we write:

```
k = 2;
```

we expect that, at run time when this statement is executed, the value 2 will be placed in that memory location reserved for the storage of the value of **k**. In C we refer to a variable such as the integer **k** as an "object".

In a sense there are two "values" associated with the object **k**. One is the value of the integer stored there (2 in the above example) and the other the "value" of the memory location, i.e., the address of **k**. Some texts refer to these two values with the nomenclature *rvalue* (right value, pronounced "are value") and *lvalue* (left value, pronounced "el value") respectively.

In some languages, the lvalue is the value permitted on the left side of the assignment operator '=' (i.e. the address where the result of evaluation of the right side ends up). The rvalue is that which is on the right side of the assignment statement, the **2** above. Rvalues cannot be used on the left side of the assignment statement. Thus: **2 = k;** is illegal.

"An *object* is a named region of storage; an *lvalue* is an expression referring to an object."

However, at this point, the definition originally cited above is sufficient. As we become more familiar with pointers we will go into more detail on this.

Okay, now consider:

```
int j, k;  
  
k = 2;  
j = 7;  <-- line 1  
k = j;  <-- line 2
```

In the above, the compiler interprets the **j** in line 1 as the address of the variable **j** (its lvalue) and creates code to copy the value 7 to that address. In line 2, however, the **j** is interpreted as its rvalue (since it is on the right hand side of the assignment operator '='). That is, here the **j** refers to the value *stored* at the memory location set aside for **j**, in this case 7. So, the 7 is copied to the address designated by the lvalue of **k**.

In all of these examples, we are using 4 byte integers so all copying of rvalues from one storage location to the other is done by copying 4 bytes. Had we been using two byte integers, we would be copying 2 bytes.

Now, let's say that we have a reason for wanting a variable designed to hold an lvalue (an address). The size required to hold such a value depends on the system. On older desk top computers with 64K of memory total, the address of any point in memory can be contained in 2 bytes. Computers with more memory would require more bytes to hold an address. The actual size required is not too important so long as we have a way of informing the compiler that what we want to store is an address.

Such a variable is called a *pointer variable* (for reasons which hopefully will become clearer a little later). In C when we define a pointer variable we do so by preceding its name with an asterisk. In C we also give our pointer a type which, in this case, refers to the type of data stored at the address we will be storing in our pointer. For example, consider the variable declaration:

```
int *ptr;
```

ptr is the name of our variable (just as **k** was the name of our integer variable). The '*' informs the compiler that we want a pointer variable, i.e. to set aside however many bytes is required to store an address in memory. The **int** says that we intend to use our pointer variable to store the address of an integer. Such a pointer is said to "point to" an integer. However, note that when we wrote **int k**; we did not give **k** a value. If this definition is made outside of any function ANSI compliant compilers will initialize it to zero. Similarly, **ptr** has no value, that is we haven't stored an address in it in the above declaration. In this case, again if the declaration is outside of any function, it is initialized to a value guaranteed in such a way that it is guaranteed to not point to any C object or function. A pointer initialized in this manner is called a "null" pointer.

The actual bit pattern used for a null pointer may or may not evaluate to zero since it depends on the specific system on which the code is developed. To make the source code compatible between various compilers on various systems, a macro is used to represent a null pointer. That macro goes under the name **NULL**. Thus, setting the value of a pointer using the **NULL** macro, as with an assignment statement such as **ptr = NULL**, guarantees that the pointer has become a null pointer. Similarly, just as one can test for an integer value of zero, as in **if(k == 0)**, we can test for a null pointer using **if (ptr == NULL)**.

But, back to using our new variable **ptr**. Suppose now that we want to store in **ptr** the address of our integer variable **k**. To do this we use the unary **&** operator and write:

```
ptr = &k;
```

What the **&** operator does is retrieve the lvalue (address) of **k**, even though **k** is on the right hand side of the assignment operator '=', and copies that to the contents of our pointer **ptr**. Now, **ptr** is said to "point to" **k**. Bear with us now, there is only one more operator we need to discuss.

The "dereferencing operator" is the asterisk and it is used as follows:

```
*ptr = 7;
```

will copy 7 to the address pointed to by **ptr**. Thus if **ptr** "points to" (contains the address of) **k**, the above statement will set the value of **k** to 7. That is, when we use the '*' this way we are referring to the value of that which **ptr** is pointing to, not the value of the pointer itself.

Similarly, we could write:

```
printf("%d", *ptr);
```

to print to the screen the integer value stored at the address pointed to by **ptr**;

One way to see how all this stuff fits together would be to run the following program and then review the code and the output carefully.

----- Program 1.1 -----


```
/* Program 1.1 from PTRTUT10.TXT 6/10/97 */
```

```
#include
```

```
int j, k;  
int *ptr;
```

```
int main(void)
```

```
{
```

```
    j = 1;
```

```
    k = 2;
```

```
    ptr = &k;
```

```
    printf("");
```

```
    printf("j has the value %d and is stored at %p", j, (void *)&j);
```

```
    printf("k has the value %d and is stored at %p", k, (void *)&k);
```

```
    printf("ptr has the value %p and is stored at %p", ptr, (void *)&ptr);
```

```
    printf("The value of the integer pointed to by ptr is %d", *ptr);
```

```
    return 0;
```

```
}
```

To review:

- A variable is declared by giving it a type and a name (e.g. **int k;**)
- A pointer variable is declared by giving it a type and a name (e.g. **int *ptr**) where the asterisk tells the compiler that the variable named **ptr** is a pointer variable and the type tells the compiler what type the pointer is to point to (integer in this case).
- Once a variable is declared, we can get its address by preceding its name with the unary **&** operator, as in **&k**.
- We can "dereference" a pointer, i.e. refer to the value of that which it points to, by using the unary ***** operator as in ***ptr**.
- An "lvalue" of a variable is the value of its address, i.e. where it is stored in memory. The "rvalue" of a variable is the value stored in that variable (at that address).

Pointer types and Arrays

Okay, let's move on. Let us consider why we need to identify the *type* of variable that a pointer points to, as in:

```
int *ptr;
```

One reason for doing this is so that later, once ptr "points to" something, if we write:

```
*ptr = 2;
```

the compiler will know how many bytes to copy into that memory location pointed to by **ptr**. If **ptr** was declared as pointing to an integer, 4 bytes would be copied.- Similarly for floats and doubles the appropriate number will be copied. But, defining the type that the pointer points to permits a number of other interesting ways a compiler can interpret code. For example, consider a block in memory consisting of ten integers in a row. That is, 40 bytes of memory are set aside to hold 10 integers.

Now, let's say we point our integer pointer **ptr** at the first of these integers. Furthermore let's say that integer is located at memory location 100 (decimal). What happens when we write:

```
ptr + 1;
```

Because the compiler "knows" this is a pointer (i.e. its value is an address) and that it points to an integer (its current address, 100, is the address of an integer), it adds 4 to **ptr** instead of 1, so the pointer "points to" the **next integer**, at memory location 104. Similarly, were the **ptr** declared as a pointer to a short, it would add 2 to it instead of 1. The same goes for other data types such as floats, doubles, or even user defined data types such as structures. This is obviously not the same kind of "addition" that we normally think of. In C it is referred to as addition using "pointer arithmetic", a term which we will come back to later.

Similarly, since **++ptr** and **ptr++** are both equivalent to **ptr + 1** (though the point in the program when **ptr** is incremented may be different), incrementing a pointer using the unary ++ operator, either pre- or post-, increments the address it stores by the amount `sizeof(type)` where "type" is the type of the object pointed to. (i.e. 4 for an integer).

Since a block of 10 integers located contiguously in memory is, by definition, an array of integers, this brings up an interesting relationship between arrays and pointers.

Consider the following:

```
int my_array[] = {1,23,17,4,-5,100};
```

Here we have an array containing 6 integers. We refer to each of these integers by means of a subscript to **my_array**, i.e. using **my_array[0]** through **my_array[5]**. But, we could alternatively access them via a pointer as follows:

```
int *ptr;
ptr = &my_array[0];    /* point our pointer at the first
                        integer in our array */
```

And then we could print out our array either using the array notation or by dereferencing our pointer. The following code illustrates this:

```
----- Program 2.1 -----
```

```
/* Program 2.1 from PTRTUT10.HTM 6/13/97 */
```

```

#include <stdio.h>

int my_array[] = {1,23,17,4,-5,100};
int *ptr;

int main(void)
{
    int i;
    ptr = &my_array[0];    /* point our pointer to the first
                           element of the array */

    printf("");
    for (i = 0; i < 6; i++)
    {
        printf("my_array[%d] = %d ",i,my_array[i]); /*<-- A */
        printf("ptr + %d = %d",i, *(ptr + i));      /*<-- B */
    }
    return 0;
}

```

Compile and run the above program and carefully note lines A and B and that the program prints out the same values in either case. Also observe how we dereferenced our pointer in line B, i.e. we first added i to it and then dereferenced the new pointer. Change line B to read:

```
printf("ptr + %d = %d",i, *ptr++);
```

and run it again... then change it to:

```
printf("ptr + %d = %d",i, *(++ptr));
```

and try once more. Each time try and predict the outcome and carefully look at the actual outcome.

In C, the standard states that wherever we might use **&var_name[0]** we can replace that with **var_name**, thus in our code where we wrote:

```
ptr = &my_array[0];
```

we can write:

```
ptr = my_array;
```

to achieve the same result.

This leads many texts to state that the name of an array is a pointer. I prefer to mentally think "the name of the array is the address of first element in the array". Many beginners (including myself when I was learning) have a tendency to become confused by thinking of it as a pointer. For example, while we can write

```
ptr = my_array;
```

we cannot write

```
my_array = ptr;
```

The reason is that while **ptr** is a variable, **my_array** is a constant. That is, the location at which the first element of **my_array** will be stored cannot be changed once **my_array[]** has been declared.

Earlier when discussing the term "lvalue" I cited K&R-2 where it stated:

"An **object** is a named region of storage; an **lvalue** is an expression referring to an object".

This raises an interesting problem. Since **my_array** is a named region of storage, why is **my_array** in the above assignment statement not an lvalue? To resolve this problem, some refer to **my_array** as an "unmodifiable lvalue".

Modify the example program above by changing

```
ptr = &my_array[0];
```

to

```
ptr = my_array;
```

and run it again to verify the results are identical.

Now, let's delve a little further into the difference between the names **ptr** and **my_array** as used above. Some writers will refer to an array's name as a **constant** pointer. What do we mean by that? Well, to understand the term "constant" in this sense, let's go back to our definition of the term "variable". When we declare a variable we set aside a spot in memory to hold the value of the appropriate type. Once that is done the name of the variable can be interpreted in one of two ways. When used on the left side of the assignment operator, the compiler interprets it as the memory location to which to move that value resulting from evaluation of the right side of the assignment operator. But, when used on the right side of the assignment operator, the name of a variable is interpreted to mean the contents stored at that memory address set aside to hold the value of that variable.

With that in mind, let's now consider the simplest of constants, as in:

```
int i, k;  
i = 2;
```

Here, while **i** is a variable and then occupies space in the data portion of memory, **2** is a constant and, as such, instead of setting aside memory in the data segment, it is imbedded directly in the code segment of memory. That is, while writing something like **k = i;** tells the compiler to create code which at run time will look at memory location **&i** to determine the value to be moved to **k**, code created by **i = 2;** simply puts the **2** in the code and there is no referencing of the data segment. That is, both **k** and **i** are objects, but **2** is not an object.

Similarly, in the above, since **my_array** is a constant, once the compiler establishes where the array itself is to be stored, it "knows" the address of **my_array[0]** and on seeing:

```
ptr = my_array;
```

it simply uses this address as a constant in the code segment and there is no referencing of the data segment beyond that.

This might be a good place explain further the use of the **(void *)** expression used in Program 1.1 of Chapter 1. As we have seen we can have pointers of various types. So far we have discussed pointers to integers and pointers to characters. In coming chapters we will be learning about pointers to structures and even pointer to pointers.

Also we have learned that on different systems the size of a pointer can vary. As it turns out it is also possible that the size of a pointer can vary depending on the data type of the object to which it points. Thus, as with integers where you can run into trouble attempting to assign a long integer to a variable of type short integer, you can run into trouble attempting to assign the values of pointers of various types to pointer variables of other types.

To minimize this problem, C provides for a pointer of type void. We can declare such a pointer by writing:

```
void *vptr;
```

A void pointer is sort of a generic pointer. For example, while C will not permit the comparison of a pointer to type integer with a pointer to type character, for example, either of these can be compared to a void pointer. Of course, as with other variables, casts can be used to convert from one type of pointer to another under the proper circumstances. In Program 1.1. of Chapter 1 I cast the pointers to integers into void pointers to make them compatible with the **%p** conversion specification. In later chapters other casts will be made for reasons defined therein.

Pointers and Strings

The study of strings is useful to further tie in the relationship between pointers and arrays. It also makes it easy to illustrate how some of the standard C string functions can be implemented. Finally it illustrates how and when pointers can and should be passed to functions.

In C, strings are arrays of characters. This is not necessarily true in other languages. In BASIC, Pascal, Fortran and various other languages, a string has its own data type. But in C it does not. In C a string is an array of characters terminated with a binary zero character (written as '\0'). To start off our discussion we will write some code which, while preferred for illustrative purposes, you would probably never write in an actual program. Consider, for example:

```
char my_string[40];

my_string[0] = 'T';
my_string[1] = 'e';
my_string[2] = 'd';
my_string[3] = '\0';
```

While one would never build a string like this, the end result is a string in that it is an array of characters **terminated with a nul character**. By definition, in C, a string is an array of characters terminated with the nul character. Be aware that "nul" is **not** the same as "NULL". The nul refers to a zero as defined by the escape sequence '\0'. That is, it occupies one byte of memory. NULL, on the other hand, is the name of the macro used to initialize null pointers. NULL is #defined in a header file in your C compiler, nul may not be #defined at all.

Since writing the above code would be very time consuming, C permits two alternate ways of achieving the same thing. First, one might write:

```
char my_string[40] = {'T', 'e', 'd', '\0'};
```

But this also takes more typing than is convenient. So, C permits:

```
char my_string[40] = "Ted";
```

When the double quotes are used, instead of the single quotes as was done in the previous examples, the nul character ('\0') is automatically appended to the end of the string.

In all of the above cases, the same thing happens. The compiler sets aside a contiguous block of memory 40 bytes long to hold characters and initializes it such that the first 4 characters are **Ted**.

Now, consider the following program:

```
-----program 3.1-----
```

```
/* Program 3.1 from PTRTUT10.HTM 6/13/97 */
```

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

```
char strA[80] = "A string to be used for demonstration purposes";
char strB[80];
```

```

int main(void)
{
    char *pA; /* a pointer to type character */
    char *pB; /* another pointer to type character */
    puts(strA); /* show string A */
    pA = strA; /* point pA at string A */
    puts(pA); /* show what pA is pointing to */
    pB = strB; /* point pB at string B */
    putchar('
'); /* move down one line on the screen */
    while(*pA != ") /* line A (see text) */
    {
        *pB++ = *pA++; /* line B (see text) */
    }
    *pB = "; /* line C (see text) */
    puts(strB); /* show strB on screen */
    return 0;
}

```

----- end program 3.1 -----

In the above we start out by defining two character arrays of 80 characters each. Since these are globally defined, they are initialized to all 0's first. Then, **strA** has the first 42 characters initialized to the string in quotes.

Now, moving into the code, we declare two character pointers and show the string on the screen. We then "point" the pointer **pA** at **strA**. That is, by means of the assignment statement we copy the address of **strA[0]** into our variable **pA**. We now use **puts()** to show that which is pointed to by **pA** on the screen. Consider here that the function prototype for **puts()** is:

```
int puts(const char *s);
```

For the moment, ignore the **const**. The parameter passed to **puts()** is a pointer, that is the **value** of a pointer (since all parameters in C are passed by value), and the value of a pointer is the address to which it points, or, simply, an address. Thus when we write **puts(strA)**; as we have seen, we are passing the address of **strA[0]**.

Similarly, when we write **puts(pA)**; we are passing the same address, since we have set **pA = strA**;

Given that, follow the code down to the **while()** statement on line A. Line A states:

While the character pointed to by **pA** (i.e. ***pA**) is not a nul character (i.e. the terminating **"**), do the following:

Line B states: copy the character pointed to by **pA** to the space pointed to by **pB**, then increment **pA** so it points to the next character and **pB** so it points to the next space.

When we have copied the last character, **pA** now points to the terminating nul character and the loop ends. However, we have not copied the nul character. And, by definition a string in C **must** be nul terminated. So, we add the nul character with line C.

It is very educational to run this program with your debugger while watching **strA**, **strB**, **pA** and **pB** and single stepping through the program. It is even more educational if instead of simply defining **strB[]** as has been done above, initialize it also with something like:

```
strB[80] = "12345678901234567890123456789012345678901234567890"
```

where the number of digits used is greater than the length of **strA** and then repeat the single stepping procedure while watching the above variables. Give these things a try!

Getting back to the prototype for **puts()** for a moment, the **"const"** used as a parameter modifier informs the user that the function will not modify the string pointed to by **s**, i.e. it will treat that string as a constant.

At this point you might want to experiment a bit with writing some of your own programs using pointers. Manipulating strings is a good place to experiment. You might want to write your own versions of such standard functions as:

```
strlen();  
strcpy();  
strchr();
```

and any others you might have on your system.

Pointers and Structures

As you may know, we can declare the form of a block of data containing different data types by means of a structure declaration. For example, a personnel file might contain structures which look something like:

```
struct tag {  
    char lname[20];    /* last name */  
    char fname[20];    /* first name */  
    int age;           /* age */  
    float rate;        /* e.g. 12.75 per hour */  
};
```


Let's say we have a bunch of these structures in a disk file and we want to read each one out and print out the first and last name of each one so that we can have a list of the people in our files. The remaining information will not be printed out. We will want to do this printing with a function call and pass to that function a pointer to the structure at hand. For demonstration purposes I will use only one structure for now. But realize the goal is the writing of the function, not the reading of the file which, presumably, we know how to do.

For review, recall that we can access structure members with the dot operator as in:

----- program 5.1 -----

/* Program 5.1 from PTRTUT10.HTM 6/13/97 */

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

```
struct tag {
    char lname[20];    /* last name */
    char fname[20];    /* first name */
    int age;           /* age */
    float rate;        /* e.g. 12.75 per hour */
};
```

```
struct tag my_struct;    /* declare the structure my_struct */
```

```
int main(void)
{
    strcpy(my_struct.lname,"Jensen");
    strcpy(my_struct.fname,"Ted");
    printf("%s ",my_struct.fname);
    printf("%s",my_struct.lname);
    return 0;
}
```

----- end of program 5.1 -----

Now, this particular structure is rather small compared to many used in C programs. To the above we might want to add:

```
    date_of_hire;           (data types not shown)
    date_of_last_raise;
    last_percent_increase;
    emergency_phone;
    medical_plan;
    Social_S_Nbr;
```

etc.....

If we have a large number of employees, what we want to do is manipulate the data in these structures by means of functions. For example we might want a function print out the name of the employee listed in any structure passed to it. However, in the original C (Kernighan & Ritchie, 1st Edition) it was not possible to pass a structure, only a pointer to a structure could be passed. In ANSI C, it is now permissible to pass the complete structure. But, since our goal here is to learn more about pointers, we won't pursue that.

Anyway, if we pass the whole structure it means that we must copy the contents of the structure from the calling function to the called function. In systems using stacks, this is done by pushing the contents of the structure on the stack. With large structures this could prove to be a problem. However, passing a pointer uses a minimum amount of stack space.

In any case, since this is a discussion of pointers, we will discuss how we go about passing a pointer to a structure and then using it within the function.

Consider the case described, i.e. we want a function that will accept as a parameter a pointer to a structure and from within that function we want to access members of the structure. For example we want to print out the name of the employee in our example structure.

Okay, so we know that our pointer is going to point to a structure declared using struct tag. We declare such a pointer with the declaration:

```
struct tag *st_ptr;
```

and we point it to our example structure with:

```
st_ptr = &my_struct;
```

Now, we can access a given member by de-referencing the pointer. But, how do we de-reference the pointer to a structure? Well, consider the fact that we might want to use the pointer to set the age of the employee. We would write:

```
(*st_ptr).age = 63;
```

Look at this carefully. It says, replace that within the parenthesis with that which **st_ptr** points to, which is the structure **my_struct**. Thus, this breaks down to the same as **my_struct.age**.

However, this is a fairly often used expression and the designers of C have created an alternate syntax with the same meaning which is:

```
st_ptr->age = 63;
```

With that in mind, look at the following program:

----- program 5.2 -----

/* Program 5.2 from PTRTUT10.HTM 6/13/97 */

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

```
struct tag{                /* the structure type */
    char lname[20];         /* last name */
    char fname[20];         /* first name */
    int age;                /* age */
    float rate;             /* e.g. 12.75 per hour */
};
```

```
struct tag my_struct;      /* define the structure */
void show_name(struct tag *p); /* function prototype */
```

```
int main(void)
{
    struct tag *st_ptr;     /* a pointer to a structure */
    st_ptr = &my_struct;    /* point the pointer to my_struct */
    strcpy(my_struct.lname, "Jensen");
    strcpy(my_struct.fname, "Ted");
    printf("%s ", my_struct.fname);
    printf("%s", my_struct.lname);
    my_struct.age = 63;
    show_name(st_ptr);      /* pass the pointer */
    return 0;
}
```

```
void show_name(struct tag *p)
{
    printf("%s ", p->fname); /* p points to a structure */
    printf("%s ", p->lname);
    printf("%d", p->age);
}
```

----- end of program 5.2 -----

Some more on Strings, and Arrays of Strings

Well, let's go back to strings for a bit. In the following all assignments are to be understood as being global, i.e. made outside of any function, including main().

We pointed out in an earlier chapter that we could write:

```
char my_string[40] = "Ted";
```

which would allocate space for a 40 byte array and put the string in the first 4 bytes (three for the characters in the quotes and a 4th to handle the terminating `'\0'`).

Actually, if all we wanted to do was store the name "Ted" we could write:

```
char my_name[] = "Ted";
```

and the compiler would count the characters, leave room for the nul character and store the total of the four characters in memory the location of which would be returned by the array name, in this case **my_name**.

In some code, instead of the above, you might see:

```
char *my_name = "Ted";
```

which is an alternate approach. Is there a difference between these? The answer is.. yes. Using the array notation 4 bytes of storage in the static memory block are taken up, one for each character and one for the terminating nul character. But, in the pointer notation the same 4 bytes required, **plus** N bytes to store the pointer variable **my_name** (where N depends on the system but is usually a minimum of 2 bytes and can be 4 or more).

In the array notation, **my_name** is short for **&my_name[0]** which is the address of the first element of the array. Since the location of the array is fixed during run time, this is a constant (not a variable). In the pointer notation **my_name** is a variable. As to which is the **better** method, that depends on what you are going to do within the rest of the program.

Let's now go one step further and consider what happens if each of these declarations are done within a function as opposed to globally outside the bounds of any function.

```
void my_function_A(char *ptr)
{
    char a[] = "ABCDEF"
    .
    .
}

void my_function_B(char *ptr)
{
    char *cp = "FGHIJ"
    .
    .
}
```

In the case of **my_function_A**, the content, or value(s), of the array **a[]** is considered to be the data. The array is said to be initialized to the values **ABCDE**. In the case of **my_function_B**, the value of the pointer **cp** is considered to be the data. The pointer has been initialized to point to the string **FGHIJ**. In both **my_function_A** and **my_function_B** the definitions are local variables and thus the string **ABCDE** is stored on the stack, as is the value of the pointer **cp**. The string **FGHIJ** can be stored anywhere. On my system it gets stored in the data segment.

Pointers to Arrays

Pointers, of course, can be "pointed at" any type of data object, including arrays. While that was evident when we discussed program 3.1, it is important to expand on how we do this when it comes to multi-dimensional arrays.

To review, in Chapter 2 we stated that given an array of integers we could point an integer pointer at that array using:

```
int *ptr;
ptr = &my_array[0];    /* point our pointer at the first
                        integer in our array */
```

As we stated there, the type of the pointer variable must match the type of the first element of the array.

In addition, we can use a pointer as a formal parameter of a function which is designed to manipulate an array. e.g.

Given:

```
int array[3] = {'1', '5', '7'};
void a_func(int *p);
```

Some programmers might prefer to write the function prototype as:

```
void a_func(int p[]);
```

which would tend to inform others who might use this function that the function is designed to manipulate the elements of an array. Of course, in either case, what actually gets passed is the value of a pointer to the first element of the array, independent of which notation is used in the function prototype or definition. Note that if the array notation is used, there is no need to pass the actual dimension of the array since we are not passing the whole array, only the address to the first element.

We now turn to the problem of the 2 dimensional array. As stated in the last chapter, C interprets a 2 dimensional array as an array of one dimensional arrays. That being the case, the first element of a 2 dimensional array of integers is a one dimensional array of integers. And a pointer to a two dimensional array of integers must be a pointer to that data type. One way of accomplishing this is through the use of the keyword "typedef". typedef assigns a new name to a specified data type. For example:

```
typedef unsigned char byte;
```

causes the name **byte** to mean type **unsigned char**. Hence

```
byte b[10];
```

 would be an array of unsigned characters.

Note that in the typedef declaration, the word **byte** has replaced that which would normally be the name of our **unsigned char**. That is, the rule for using **typedef** is that the new name for the data type is the name used in the definition of the data type. Thus in:

```
typedef int Array[10];
```

Array becomes a data type for an array of 10 integers. i.e. **Array my_arr;** declares **my_arr** as an array of 10 integers and **Array arr2d[5];** makes **arr2d** an array of 5 arrays of 10 integers each.

Also note that **Array *p1d;** makes **p1d** a pointer to an array of 10 integers. Because ***p1d** points to the same type as **arr2d**, assigning the address of the two dimensional array **arr2d** to **p1d**, the pointer to a one dimensional array of 10 integers is acceptable. i.e. **p1d = &arr2d[0];** or **p1d = arr2d;** are both correct.

Since the data type we use for our pointer is an array of 10 integers we would expect that incrementing **p1d** by 1 would change its value by **10*sizeof(int)**, which it does. That is, **sizeof(*p1d)** is 20. You can prove this to yourself by writing and running a simple short program.

Now, while using typedef makes things clearer for the reader and easier on the programmer, it is not really necessary. What we need is a way of declaring a pointer like **p1d** without the need of the **typedef** keyword. It turns out that this can be done and that

```
int (*p1d)[10];
```

is the proper declaration, i.e. **p1d** here is a pointer to an array of 10 integers just as it was under the declaration using the **Array** type. Note that this is different from

```
int *p1d[10];
```

which would make **p1d** the name of an array of 10 pointers to type **int**.

Pointers and Dynamic Allocation of Memory

There are times when it is convenient to allocate memory at run time using **malloc()**, **calloc()**, or other allocation functions. Using this approach permits postponing the decision on the size of the memory block need to store an array, for example, until run time. Or it permits using a section of memory for the storage of an array of integers at one point in time, and then when that memory is no longer needed it can be freed up for other uses, such as the storage of an array of structures.

When memory is allocated, the allocating function (such as **malloc()**, **calloc()**, etc.) returns a pointer. The type of this pointer depends on whether you are using an older K&R compiler or the

newer ANSI type compiler. With the older compiler the type of the returned pointer is **char**, with the ANSI compiler it is **void**.

If you are using an older compiler, and you want to allocate memory for an array of integers you will have to cast the char pointer returned to an integer pointer. For example, to allocate space for 10 integers we might write:

```
int *iptr;  
iptr = (int *)malloc(10 * sizeof(int));  
if (iptr == NULL)
```

```
{ .. ERROR ROUTINE GOES HERE .. }
```

If you are using an ANSI compliant compiler, **malloc()** returns a **void** pointer and since a void pointer can be assigned to a pointer variable of any object type, the **(int *)** cast shown above is not needed. The array dimension can be determined at run time and is not needed at compile time. That is, the **10** above could be a variable read in from a data file or keyboard, or calculated based on some need, at run time.

Because of the equivalence between array and pointer notation, once **iptr** has been assigned as above, one can use the array notation. For example, one could write:

```
int k;  
for (k = 0; k < 10; k++)  
    iptr[k] = 2;  
to set the values of all elements to 2.
```

Even with a reasonably good understanding of pointers and arrays, one place the newcomer to C is likely to stumble at first is in the dynamic allocation of multi-dimensional arrays. In general, we would like to be able to access elements of such arrays using array notation, not pointer notation, wherever possible. Depending on the application we may or may not know both dimensions at compile time. This leads to a variety of ways to go about our task.

As we have seen, when dynamically allocating a one dimensional array its dimension can be determined at run time. Now, when using dynamic allocation of higher order arrays, we never need to know the first dimension at compile time. Whether we need to know the higher dimensions depends on how we go about writing the code.

Pointers to Functions

Up to this point we have been discussing pointers to data objects. C also permits the declaration of pointers to functions. Pointers to functions have a variety of uses and some of them will be discussed here.

Consider the following real problem. You want to write a function that is capable of sorting virtually any collection of data that can be stored in an array. This might be an array of strings, or integers, or floats, or even structures. The sorting algorithm can be the same for all. For example,

it could be a simple bubble sort algorithm, or the more complex shell or quick sort algorithm. We'll use a simple bubble sort for demonstration purposes.

Sedgewick [1] has described the bubble sort using C code by setting up a function which when passed a pointer to the array would sort it. If we call that function **bubble()**, a sort program is described by bubble_1.c, which follows:

```
/*----- bubble_1.c -----*/

/* Program bubble_1.c from PTRTUT10.HTM 6/13/97 */

#include <stdio.h>

int arr[10] = { 3,6,1,2,3,8,4,1,7,2};

void bubble(int a[], int N);

int main(void)
{
    int i;
    putchar("");
    for (i = 0; i < 10; i++)
    {
        printf("%d ", arr[i]);
    }
    bubble(arr,10);
    putchar("");

    for (i = 0; i < 10; i++)
    {
        printf("%d ", arr[i]);
    }
    return 0;
}

void bubble(int a[], int N)
{
    int i, j, t;
    for (i = N-1; i >= 0; i--)
    {
        for (j = 1; j <= i; j++)
        {
            if (a[j-1] > a[j])
            {
                t = a[j-1];
                a[j-1] = a[j];
            }
        }
    }
}
```



```

        a[j] = t;
    }
}
}
}

```

```
/*----- end bubble_1.c -----*/
```

The bubble sort is one of the simpler sorts. The algorithm scans the array from the second to the last element comparing each element with the one which precedes it. If the one that precedes it is larger than the current element, the two are swapped so the larger one is closer to the end of the array. On the first pass, this results in the largest element ending up at the end of the array. The array is now limited to all elements except the last and the process repeated. This puts the next largest element at a point preceding the largest element. The process is repeated for a number of times equal to the number of elements minus 1. The end result is a sorted array.

Here our function is designed to sort an array of integers. Thus in line 1 we are comparing integers and in lines 2 through 4 we are using temporary integer storage to store integers. What we want to do now is see if we can convert this code so we can use any data type, i.e. not be restricted to integers.

At the same time we don't want to have to analyze our algorithm and the code associated with it each time we use it. We start by removing the comparison from within the function **bubble()** so as to make it relatively easy to modify the comparison function without having to re-write portions related to the actual algorithm. This results in bubble_2.c:

```
/*----- bubble_2.c -----*/
```

```
/* Program bubble_2.c from PTB/TUT10.HTM 6/13/97 */
```

```
/* Separating the comparison function */
```

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

```
int arr[10] = { 3,6,1,2,3,8,4,1,7,2};
```

```
void bubble(int a[], int N);
```

```
int compare(int m, int n);
```

```
int main(void)
```

```
{
```

```
    int i;
```

```
    putchar("");
```

```
    for (i = 0; i < 10; i++)
```

```

    {
        printf("%d ", arr[i]);
    }
    bubble(arr,10);
    putchar("\n");

    for (i = 0; i < 10; i++)
    {
        printf("%d ", arr[i]);
    }
    return 0;
}

```

```
void bubble(int a[], int N)
```

```

{
    int i, j, t;
    for (i = N-1; i >= 0; i--)
    {
        for (j = 1; j <= i; j++)
        {
            if (compare(a[j-1], a[j]))
            {
                t = a[j-1];
                a[j-1] = a[j];
                a[j] = t;
            }
        }
    }
}

```

```
int compare(int m, int n)
```

```

{
    return (m > n);
}

```

```
/*----- end of bubble_2.c -----*/
```

If our goal is to make our sort routine data type independent, one way of doing this is to use pointers to type void to point to the data instead of using the integer data type. As a start in that direction let's modify a few things in the above so that pointers can be used. To begin with, we'll stick with pointers to type integer.

```
/*----- bubble_3.c -----*/
```

```
/* Program bubble_3.c from PTRTUT10.HTM 6/13/97 */
```

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

```
int arr[10] = { 3,6,1,2,3,8,4,1,7,2};
```

```
void bubble(int *p, int N);  
int compare(int *m, int *n);
```

```
int main(void)  
{  
    int i;  
    putchar("");  
  
    for (i = 0; i < 10; i++)  
    {  
        printf("%d ", arr[i]);  
    }  
    bubble(arr,10);  
    putchar("");  
  
    for (i = 0; i < 10; i++)  
    {  
        printf("%d ", arr[i]);  
    }  
    return 0;  
}
```

```
void bubble(int *p, int N)  
{  
    int i, j, t;  
    for (i = N-1; i >= 0; i--)  
    {  
        for (j = 1; j <= i; j++)  
        {  
            if (compare(&p[j-1], &p[j]))  
            {  
                t = p[j-1];  
                p[j-1] = p[j];  
                p[j] = t;  
            }  
        }  
    }  
}
```

```
int compare(int *m, int *n)  
{  
    return (*m > *n);  
}
```

```
/*----- end of bubble3.c -----*/
```

Note the changes. We are now passing a pointer to an integer (or array of integers) to **bubble()**. And from within bubble we are passing pointers to the elements of the array that we want to compare to our comparison function. And, of course we are dereferencing these pointer in our **compare()** function in order to make the actual comparison. Our next step will be to convert the pointers in **bubble()** to pointers to type void so that that function will become more type insensitive. This is shown in bubble_4.

```
/*----- bubble_4.c -----*/
```

```
/* Program bubble_4.c from PTRTUT10,HTM 6/13/97 */
```

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

```
int arr[10] = { 3,6,1,2,3,8,4,1,7,2};
```

```
void bubble(int *p, int N);  
int compare(void *m, void *n);
```

```
int main(void)
```

```
{  
    int i;  
    putchar("");  
  
    for (i = 0; i < 10; i++)  
    {  
        printf("%d ", arr[i]);  
    }  
    bubble(arr,10);  
    putchar("");
```

```
    for (i = 0; i < 10; i++)  
    {  
        printf("%d ", arr[i]);  
    }  
    return 0;
```

```
}
```

```
void bubble(int *p, int N)
```

```
{  
    int i, j, t;  
    for (i = N-1; i >= 0; i--)  
    {  
        for (j = 1; j <= i; j++)
```

```

    {
        if (compare((void *)&p[j-1], (void *)&p[j]))
        {
            t = p[j-1];
            p[j-1] = p[j];
            p[j] = t;
        }
    }
}

int compare(void *m, void *n)
{
    int *m1, *n1;
    m1 = (int *)m;
    n1 = (int *)n;
    return (*m1 > *n1);
}

/*----- end of bubble_4.c -----*/

```

Note that, in doing this, in **compare()** we had to introduce the casting of the void pointer types passed to the actual type being sorted. But, as we'll see later that's okay. And since what is being passed to **bubble()** is still a pointer to an array of integers, we had to cast these pointers to void pointers when we passed them as parameters in our call to **compare()**.

We now address the problem of what we pass to **bubble()**. We want to make the first parameter of that function a void pointer also. But that means that within **bubble()** we need to do something about the variable **t**, which is currently an integer. Also, where we use **t = p[j-1]**; the type of **p[j-1]** needs to be known in order to know how many bytes to copy to the variable **t** (or whatever we replace **t** with)

Currently, in **bubble_4.c**, knowledge within **bubble()** as to the type of the data being sorted (and hence the size of each individual element) is obtained from the fact that the first parameter is a pointer to type integer. If we are going to be able to use **bubble()** to sort any type of data, we need to make that pointer a pointer to type **void**. But, in doing so we are going to lose information concerning the size of individual elements within the array. So, in **bubble_5.c** we will add a separate parameter to handle this size information.

File Handling in C

We frequently use files for storing information which can be processed by our programs. In order to store information permanently and retrieve it we need to use files. Files are not only used for data. Our programs are also stored in files. The editor which you use to enter your program and save it, simply manipulates files for you.

The Unix commands cat, cp, cmp are all programs which process your files. In order to use files we have to learn about **File I/O** i.e. how to write information to a file and how to read information from a file. We will see that file I/O is almost identical to the terminal I/O that we have been using so far. The primary difference between manipulating files and doing terminal I/O is that we must specify in our programs which files we wish to use.

As you know, you can have many files on your disk. If you wish to use a file in your programs, then you must specify which file or files you wish to use. Specifying the file you wish to use is referred to as **opening** the file.

When you open a file you must also specify what you wish to do with it i.e. **Read** from the file, **Write** to the file, or both. Because you may use a number of different files in your program, you must specify when reading or writing which file you wish to use. This is accomplished by using a variable called a **file pointer**. Every file you open has its own file pointer variable. When you wish to write to a file you specify the file by using its file pointer variable. You declare these file pointer variables as follows:

FILE *fopen(), *fp1, *fp2, *fp3;

The variables fp1, fp2, fp3 are file pointers. You may use any name you wish.

The file <stdio.h> contains declarations for the Standard I/O library and should always be **included** at the very beginning of C programs using files. Constants such as FILE, EOF and NULL are defined in <stdio.h>.

You should note that a file pointer is simply a variable like an integer or character. It does **not** point to a file or the data in a file. It is simply used to indicate which file your I/O operation refers to. A file number is used in the Basic language and a unit number is used in Fortran for the same purpose. The function **fopen** is one of the Standard Library functions and returns a file pointer which you use to refer to the file you have opened e.g.

```
fp = fopen( "prog.c", "r" );
```

The above statement **opens** a file called prog.c for **reading** and associates the file pointer fp with the file.

When we wish to access this file for I/O, we use the file pointer variable fp to refer to it.

You can have up to about 20 files open in your program - you need one file pointer for each file you intend to use.

File I/O

The Standard I/O Library provides similar routines for file I/O to those used for standard I/O. The routine getc(fp) is similar to getchar() and putc(c,fp) is similar to putchar(c).

Thus the statement

```
c = getc(fp);
```

reads the next character from the file referenced by fp and the statement

```
    putc(c,fp);
```

writes the character `c` into file referenced by `fp`.

```
/* file.c: Display contents of a file on screen */
```

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

```
void main()
```

```
{
```

```
    FILE *fopen(), *fp;
```

```
    int c ;
```

```
    fp = fopen( "prog.c", "r" );
```

```
    c = getc( fp ) ;
```

```
    while ( c != EOF )
```

```
    {
```

```
        putchar( c );
```

```
        c = getc ( fp );
```

```
    }
```

```
    fclose( fp );
```

```
}
```

In this program, we open the file `prog.c` for reading.

We then read a character from the file. This file must exist for this program to work.

If the file is empty, we are at the end, so `getc` returns `EOF` a special value to indicate that the end of file has been reached. (Normally `-1` is used for `EOF`)

The while loop simply keeps reading characters from the file and displaying them, until the end of the file is reached.

The function **`fclose`** is used to ***close*** the file i.e. indicate that we are finished processing this file.

We could reuse the file pointer `fp` by opening another file.

This program is in effect a special purpose `cat` command. It displays file contents on the screen, but **only** for a file called `prog.c`.

By allowing the user enter a file name, which would be stored in a string, we can modify the above to make it an **interactive** cat command:

```
/* cat2.c: Prompt user for filename and display file on screen */
```

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

```
void main()
```

```
{
```

```
    FILE *fopen(), *fp;
```

```
    int c ;
```

```
    char filename[40] ;
```

```
    printf("Enter file to be displayed: ");
```

```
    gets( filename ) ;
```

```
    fp = fopen( filename, "r");
```

```
    c = getc( fp ) ;
```

```
    while ( c != EOF )
```

```
    {
```

```
        putchar(c);
```

```
        c = getc ( fp );
```

```
    }
```

```
    fclose( fp );
```

```
}
```

In this program, we pass the name of the file to be opened which is stored in the array called filename, to the fopen function. In general, anywhere a string constant such as "prog.c" can be used so can a character array such as filename. (Note the **reverse** is **not** true).

The above programs suffer a major limitation. They **do not** check whether the files to be used exist or not.

If you attempt to read from a non-existent file, your program will crash!!

The `fopen` function was designed to cope with this eventuality. It checks if the file can be opened appropriately. If the file **cannot be opened**, it returns a **NULL** pointer. Thus by checking the file pointer returned by `fopen`, you can determine if the file was opened correctly and take appropriate action e.g.

```
fp = fopen (filename, "r") ;

if ( fp == NULL)

{

    printf("Cannot open %s for reading \n", filename );

    exit(1) ; /*Terminate program: Commit suicide !!*/

}
```

The above code fragment show how a program might check if a file could be opened appropriately.

The function **exit()** is a special function which terminates your program immediately.

`exit(0)` mean that you wish to indicate that your program terminated successfully whereas a nonzero value means that your program is terminating due to an error condition.

Alternatively, you could prompt the user to enter the filename again, and try to open it again:

```
fp = fopen (fname, "r") ;

while ( fp == NULL)

{

    printf("Cannot open %s for reading \n", fname );

    printf("\n\nEnter filename :") ;

    gets( fname );

    fp = fopen (fname, "r") ;

}
```

In this code fragment, we keep reading filenames from the user until a valid existing filename is entered.

Writing to Files

The previous programs have opened files for reading and read characters from them.

To write to a file, the file must be opened for writing e.g.

```
fp = fopen( fname, "w" );
```

If the file does not exist already, it will be created. If the file does exist, it will be overwritten! So, be careful when opening files for writing, in case you destroy a file unintentionally. Opening files for writing can also fail. If you try to create a file in another users directory where you do not have access you will not be allowed and fopen will fail.

Character Output to Files

The function `putc(c, fp)` writes a character to the file associated with the file pointer `fp`.

Example:

Write a file copy program which copies the file “prog.c” to “prog.old”

Outline solution:

Open files appropriately

Check open succeeded

Read characters from prog.c and

Write characters to prog.old until all characters copied

Close files

The step: “Read characters and write ..” may be refined to:

```
read character from prog.c
while not end of file do
begin
    write character to prog.old
    read next character from prog.c
end
```

```
/* filecopy.c : Copy prog.c to prog.old */
```

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

```
void main()
```

```
{
```

```

FILE *fp1, *fp2, *fopen();

int c ;

fp1 = fopen( "prog.c", "r" );    /* open for reading */

fp2 = fopen( "prog.old", "w" ) ; /* open for writing */

if ( fp1 == NULL )    /* check does file exist etc */
{
    printf("Cannot open prog.c for reading \n" );
    exit(1); /* terminate program */
}

else if ( fp2 == NULL )
{
    printf("Cannot open prog.old for writing \n"),
    exit(1); /* terminate program */
}

else    /* both files O.K. */
{
    c = getc(fp1) ;
    while ( c != EOF )
    {
        putc( c, fp2); /* copy to prog.old */
        c = getc( fp1 ) ;
    }

    fclose ( fp1 );    /* Now close files */
    fclose ( fp2 );
    printf("Files successfully copied \n");
}

```

```
}
```

The above program only copies the specific file prog.c to the file prog.old. We can make it a general purpose program by prompting the user for the files to be copied and opening them appropriately.

```
/* copy.c : Copy any user file*/
```

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

```
void main()
```

```
{
```

```
    FILE *fp1, *fp2, *fopen();
```

```
    int c ;
```

```
    char fname1[40], fname2[40] ;
```

```
    printf("Enter source file:");
```

```
    gets(fname1);
```

```
    printf("Enter destination file:");
```

```
    gets(fname2);
```

```
    fp1 = fopen( fname1, "r" );    /* open for reading */
```

```
    fp2 = fopen( fname2, "w" );    /* open for writing */
```

```
    if ( fp1 == NULL )    /* check does file exist etc */
```

```
    {
```

```
        printf("Cannot open %s for reading \n", fname1 );
```

```
        exit(1);    /* terminate program */
```

```
    }
```

```
    else if ( fp2 == NULL )
```

```
    {
```

```
        printf("Cannot open %s for writing \n", fname2 );
```

```
        exit(1);    /* terminate program */
```

```
    }
```

```

else          /* both files O.K. */
{
    c = getc(fp1);          /* read from source */
    while ( c != EOF)
    {
        putc( c, fp2);  /* copy to destination */
        c = getc( fp1 );
    }

    fclose ( fp1 );          /* Now close files */
    fclose ( fp2 );
    printf("Files successfully copied \n");
}
}

```

Command Line Parameters: Arguments to main()

Accessing the command line arguments is a very useful facility. It enables you to provide commands with arguments that the command can use e.g. the command

```
% cat prog.c
```

takes the argument "prog.c" and opens a file with that name, which it then displays. The command line arguments include the command name itself so that in the above example, "cat" and "prog.c" are the command line arguments. The first argument i.e. "cat" is argument number zero, the next argument, "prog.c", is argument number one and so on.

To access these arguments from within a C program, you pass parameters to the function main(). The use of arguments to main is a key feature of many C programs.

The declaration of main looks like this:

```
int main (int argc, char *argv[])
```

This declaration states that

1. main returns an integer value (used to determine if the program terminates successfully)
2. argc is the number of command line arguments including the command itself i.e argc must be at least 1
3. argv is an array of the command line arguments

The declaration of argv means that it is an array of pointers to strings (the command line arguments). By the normal rules about arguments whose type is array, what actually gets passed to main is the address of the first element of the array. As a result, an equivalent (and widely used) declaration is:

```
int main (int argc, char **argv)
```

When the program starts, the following conditions hold true:

- o argc is greater than 0.
- o argv[argc] is a null pointer.
- o argv[0], argv[1], ..., argv[argc-1] are pointers to strings with implementation defined meanings.
- o argv[0] is a string which contains the program's name, or is an empty string if the name isn't available. Remaining members of argv are the program's arguments.

Example: print_args echoes its arguments to the standard output – is a form of the Unix echo command.

```
/* print_args.c: Echo command line arguments */
```

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

```
#include <stdlib.h>
```

```
int main(int argc, char *argv[])
```

```
{
```

```
    int i = 0;
```

```
    int num_args ;
```

```
    num_args = argc ;
```

```
    while( num_args > 0)
```

```
    {
```

```
        printf("%s\n", argv[i]);
```

```

        i++;
        num_args--;
    }
}

```

If the name of this program is print_args, an example of its execution is as follows:

```
% print_args hello goodbye solong
```

```
print_args
```

```
hello
```

```
goodbye
```

```
solong
```

```
%
```

Dynamic Memory Allocation In C

Dynamic memory allocation means, a program can obtain its memory while it is running. It allows us to allocate additional memory space or to release unwanted space at the time of program execution (runtime). Dynamic memory allocation function

FUNCTION

MEANING

- malloc() Used to allocate blocks of memory in required size of bytes.
- free() Used to release previously allocated memory space.
- calloc() Used to allocate memory space for an array of elements
- realloc() Used to modify the size of the previously allocated memory space.

To use these functions <stdlib.h> header file is to be included.

Malloc() Function

The malloc() function is used to allocate block of memory (i.e.) it allocates a block of memory of specified size and return a pointer of type void.

Syntax:

```
pointer_variable=(typecast*)malloc(size of bytes)
```

Eg:

```
char *a;  
  
a=malloc(5);
```

5 bytes of memory space is reserved and the address of first byte is stored in pointer variable 'a'

Calloc() Function

The calloc() function is used for allocating memory space during the program execution for derived data types such as arrays, structures, etc.

Syntax: pointer_variable=(typecast*)calloc(n,element size)

Example

```
struct book  
{  
    int no;  
    char name[10];  
    float cost;  
};  
  
struct book b1;  
  
b1 *sptr;  
  
sptr=(book *) calloc(10, sizeof(book));
```

Realloc() Function

It is necessary to alter the previously allocated memory. i.e., to add additional memory or to reduce as and when required. For above purposes, the realloc() function is very useful and this process is called reallocation of memory. Before using this statement, the user must allocate some memory previously by using the malloc() function.

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



```
#include<stdlib.h>

main()
{
    char *p;      /* *p is a pointer variables */
    p=(char *)malloc(6);
    strcpy(p,"ROHTAK");
    printf("Memory contains:%s\n",p)
    p=(char *)realloc(p,7) /* reallocation */
    strcpy(p,"HARYANA");
    printf("Memory now contains: %s\n", p);
    free(p);/* Releasing memory*/
}
```

OUTPUT

Memory contains: ROHTAK

Memory now contains: HARYANA