Contents

1 Introduction

1.1 History of Computing and Computers

- The history of computers starts out about 2000 years ago in Babylonia (Mesopotamia), at the birth of the abacus, a wooden rack holding two horizontal wires with beads strung on them.
- Blaise Pascal is usually credited for building the first digital computer in 1642. It added numbers entered with dials and was made to help his father, a tax collector.
  The basic principle of his calculator is still used today in water meters and modern-day odometers. Instead of having a carriage wheel turn the gear, he made each ten-teeth wheel accessible to be turned directly by a person's hand (later inventors added keys and a crank), with the result that when the wheels were turned in the proper sequences, a series of numbers was entered and a cumulative sum was obtained. The gear train supplied a mechanical answer equal to the answer that is obtained by using arithmetic. This first mechanical calculator, called the Pascaline, had several disadvantages. Although it did offer a substantial improvement over manual calculations, only Pascal himself could repair the device and it cost more than the people it replaced! In addition, the first signs of technophobia emerged with mathematicians fearing the loss of their jobs due to progress.
- A step towards automated computing was the development of punched cards, which were first successfully used with computers in 1890 by Herman Hollerith and James Powers, who worked for the US. Census Bureau. They developed devices that could read the information that had been punched into the cards automatically, without human help. Because of this, reading errors were reduced dramatically, work flow increased, and, most importantly, stacks of punched cards could be used as easily accessible memory of almost unlimited size. Furthermore, different problems could be stored on different stacks of cards and accessed when needed.
- These advantages were seen by commercial companies and soon led to the development of improved punch-card using computers created by International Business Machines (IBM), Remington (yes, the same people that make shavers), Burroughs, and other corporations. These computers used electromechanical devices in which electrical power provided mechanical motion
- The start of World War II produced a large need for computer capacity, especially for the military. New weapons were made for which trajectory tables and other essential data were needed. In 1942, John P. Eckert, John W. Mauchly, and their associates at the Moore school of Electrical Engineering of University of Pennsylvania decided to build a high - speed electronic computer to do the job. This machine became known as ENIAC (Electrical Numerical Integrator and Calculator).
  The size of ENIAC′s numerical "word" was 10 decimal digits, and it could multiply two of these numbers at a rate of 300 per second, by finding the value of each product from a multiplication table stored in its memory. ENIAC was therefore about 1,000 times faster than the previous generation of relay computers. ENIAC used 18,000 vacuum tubes, about 1,800 square feet of floor space, and consumed about 180,000 watts of electrical power. It had punched card I/O, 1 multiplier, 1 divider/square rooter, and 20 adders using decimal ring counters, which served as adders and also as quick-access (.0002 seconds) read-write register storage. The executable instructions making up a program were embodied in the separate "units" of ENIAC, which were plugged together to form a "route" for the flow of information.
- Early in the 50′s two important engineering discoveries changed the image of the electronic - computer field, from one of fast but unreliable hardware to an image of relatively high reliability and even more capability. These discoveries were the magnetic core memory and the Transistor - Circuit Element.
  These technical discoveries quickly found their way into new models of digital computers. RAM capacities increased from 8,000 to 64,000 words in commercially available machines by the 1960′s, with access times of 2 to 3 MS (Milliseconds). These machines were very expensive to purchase or even to rent and were particularly expensive to operate because of the cost of expanding programming. Such computers were mostly found in large computer centers operated by industry, government, and private laboratories - staffed with many programmers and support personnel. This situation led to modes of operation enabling the sharing of the high potential available.
- Many companies, such as Apple Computer and Radio Shack, introduced very successful PC′s in the 1970's, encouraged in part by a fad in computer (video) games. In the 1980's some friction occurred in the crowded PC field, with Apple and IBM keeping strong. In the manufacturing of semiconductor chips, the Intel and Motorola Corporations were very competitive into the 1980s, although Japanese firms were making strong economic advances, especially in the area of memory chips. By the late 1980s, some personal computers were run by microprocessors that, handling 32 bits of data at a time, could process about 4,000,000 instructions per second.

1.2 Generation of Computer

A generation refers to the state of improvement in the development of a product. This term is also used in the different advancements of computer technology. With each new generation, the circuitry has gotten smaller and more advanced than the previous generation before it. As a result of the miniaturization, speed, power, and memory of computers has proportionally increased. New discoveries are constantly being developed that affect the way we live, work and play

1. **First Generation of Computers (1940-1956):Vacuum Tubes**
   The first computers used vacuum tubes for circuitry and magnetic drums for memory, and were often enormous, taking up entire rooms. They were very expensive to operate and in addition to using a great deal of electricity, the first computers generated a lot of heat, which was often the cause of malfunctions.
   First generation computers relied on machine language, the lowest-level programming language understood by computers, to perform operations, and they could only solve one problem at a time, and it could take days or weeks to set-up a new problem. Input was based on punched cards and paper tape, and output was displayed on printouts.
   The UNIVAC and ENIAC computers are examples of first-generation computing devices. The UNIVAC was the first commercial computer delivered to a business client, the U.S. Census Bureau in 1951.
2. **Second Generation of Computers (1956-1963):Transistors**
   Transistors replace vacuum tubes and ushered in the second generation of computers. The transistor was invented in 1947 but did not see widespread use in computers until the late 1950s. The transistor was far superior to the vacuum tube, allowing computers to become smaller, faster, cheaper, more energy-efficient and more reliable than their first-generation predecessors.
   Though the transistor still generated a great deal of heat that subjected the computer to damage, it was a vast improvement over the vacuum tube. Second-generation computers still relied on punched cards for input and printouts for output.
   Second-generation computers moved from cryptic binary machine language to symbolic, or assembly, languages, which allowed programmers to specify instructions in words. High-level programming languages were also being developed at this time, such as early versions of COBOL and FORTRAN. These were also the first computers that stored their instructions in their memory, which moved from a magnetic drum to magnetic core technology.
   The first computers of this generation were developed for the atomic energy industry.
3. **Third Generation of Computers(1964-1971): Integrated Circuits**
   The development of the integrated circuit was the hallmark of the third generation of computers. Transistors were miniaturized and placed on silicon chips, called semiconductors, which drastically increased the speed and efficiency of computers.
   Instead of punched cards and printouts, users interacted with third generation computers through keyboards and monitors and interfaced with an operating system, which allowed the device to run many different applications at one time with a central program that monitored the memory. Computers for the first time became accessible to a mass audience because they were smaller and cheaper than their predecessors.
4. **Fourth Generation of Computers(1971-Present): Microprocessors**
   The microprocessor brought the fourth generation of computers, as thousands of integrated circuits were built onto a single silicon chip. What in the first generation filled an entire room could now fit in the palm of the hand. The Intel 4004 chip, developed in 1971, located all the components of the computer—from the central processing unit and memory to input/output controls on a single chip.
   In 1981 IBM introduced its first computer for the home user, and in 1984 Apple introduced the Macintosh. Microprocessors also moved out of the realm of desktop computers and into many areas of life as more and more everyday products began to use microprocessors.
   As these small computers became more powerful, they could be linked together to form networks, which eventually led to the development of the Internet. Fourth generation computers also saw the development of GUIs, the mouse and handheld devices.
5. **Fifth Generation of Computers(Present and Beyond): Artificial Intelligence**
   Fifth generation computing devices, based on artificial intelligence, are still in development, though there are some applications, such as voice recognition, that are being used today. The use of parallel processing and superconductors is helping to make artificial intelligence a reality. Quantum computation and molecular and nanotechnology will radically change the face of computers in years to come. The goal of fifth-generation computing is to develop devices that respond to natural language input and are capable of learning and self-organization.

1.3 Block Diagram of Computer

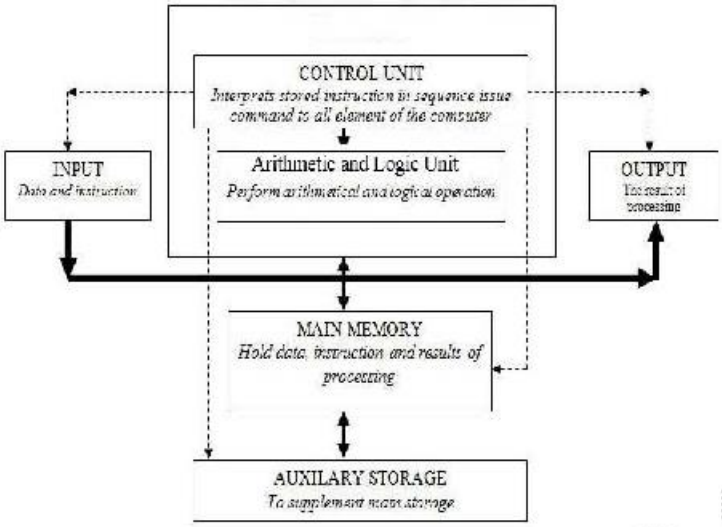


Figure 1-1: Block Diagram of Computer

Input Unit:

The process of sending the data and Instructions for the processing through some suitable devices such as Keyboard, Mouse etc. is called Input. The devices translate the data from human understandable form into electronic impulses which are understood by the computer.

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.

- **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.
- **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.

Central Processing Unit:

The Control Unit (CU) and Arithmetic and Logic Unit (ALU) of the computer are together known as the Central Processing Unit (CPU).Once the data accepted it fed in to Central Processing Unit before the output is generated as data has to be processed, which is done by CPU. This unit of the computer is the brain of computer system, which does all the processing, calculations, problem solving and controls all other functions of all other elements of the computer.

- **Arithmetic and Logical Unit (ALU):** 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 (CU):** 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 its working.

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.

1.4 Classification of Computers

1.4.1 On the basis of working mode

- **Analog Computer**
  - Analog computer measures continuous types of data and uses a physical quantity like, electric current, voltage, temperature etc.
  - To present and process the data. We have seen many Analog devices in our life.it represent numbers by a physical quantity; that is, they assign numeric values by physically measuring some actual property, such as the length of an object, an angle created by two lines, or the amount of voltage passing through a point in an electric circuit.
  - It usually contains either no any or limited storage capacity.
  - Thermometer, multimeter, speedometer, fuel and price indicator in petrol pump are the examples of Analog devices.
  - 'Presley' is an example of analog computer.
- **Digital Computer**
  - The computer which accepts discrete data (discontinuous data) as per the electric signals is known as digital computer basically, digital computer counts digits which represent numbers or letters. They are the most widely used type of computers. The computers based on binary digits i.e. 0 and 1 are called digital computer.it represents each and every information(number, letter and other special symbols) in terms of single numbers (0 and 1)and processes these information by using standard arithmetic operations.
  - It usually contains larger storage capacity.
  - Devices like digital watch, digital speedometer, etc. are the examples of digital devices.
  - IBM desktop PC, Dell laptop, Acer notebook are the examples of digital computers.
- **Hybrid Computer**
  - Hybrid computer is the combination of the features of Analog and digital computers.it has both features of Analog and digital computer.it can do all types of tasks of digital and analog.
  - It can convert analog data to digital and vice versa.
  - In a hybrid computer, analog component is used for measuring and comparing, and digital component is used for controlling.
  - Its storage capacity varies from the application area.
  - They are mostly used in scientific research, industrial application, aero planes etc.

1.4.2 On the basis of size

- **Super Computer**

  Super computer is the fastest, most expensive, big in size, and most powerful computer that can perform multiple tasks within no second. It has multi-user, multiprocessing, very high efficiency and large amount of storage capacity. It is called super computer because it can solve difficult and complex problem within a nano second.
- **Mainframe Computer**

  Mainframe Computer is the large sized computer that covers about 1000 sq feet. It is general purpose computer that is designed to process large amount of data with very high speed. It accepts large amount of data from different terminals and multiple users and process them at same time. More than 100 users are allowed to work in this system. It is applicable for large organization with multi-users for example: large business organization, Department of examinations, Industries and defense to process data of complex nature. It uses several CPU for data processing.
- **Mini Computer**

  Mini Computers are medium sized computer. So, these are popular as middle ranged computer. It is also multiple user computer and supports more than dozen of people at a time. It is costlier than microcomputer.

  It is also used in university, middle range business organizations to process complex data. It is also used in scientific research, instrumentation system, engineering analysis, and industrial process monitoring and control etc.
- **Micro Computer**

  Most popular general purpose computers which are mostly used on day to day work are microcomputers. These are popular as Home PC or Personal Computer (PC) because these are single user computers and mostly used for personal use and application. These support many higher level application cost and easy in operation.

1.5 Computer Software

Software is defined as a set of coded commands (program) that tell a computer what tasks to perform. Without software a computer cannot do anything. These program is called software because these are relatively easy to change both the instructions in a particular program as well as which program is being executed by the hardware at any given time.

1.5.1 System Software

It is a term referring to any computer software whose purpose is to help run the computer system. Most of it is responsible directly for controlling, integrating and managing the individual hardware components of a computer system. Specific kinds of system software include operating systems (OS), device drivers, programming tools etc. An operating system is a main system software, it controls all parts of the computer system. The major functions of the OS are to handle all input devices, handle all output devices, coordinate and manage use of other resources like memory, disk, CPU etc., accept commands from users, provide an environment over which other programs (software) can run, transferring data from memory to disk or rendering text onto a display etc. DOS (disk operating system), windows, Linux are some popular examples of operating system.

1.5.2 Application Software

Application software is a subclass of computer software that employs the capabilities of a computer directly to a task that the user wishes to perform. Application software are designed to process data and support users in an organization such as solving equations or producing bills, result processing of campuses, data processing of accounts in the bank etc. Most application software are designed to meet a generic set of business requirements that will suit a very large number of business customers. These are specially designed and developed for end users. These run on top of the operating system. Example include word processing software like Word/Word Perfect, spreadsheet like Excel or Lotus 123 etc. Application software can be classified into following categories:

- **Tailored Software:** These kinds of software are developed for solving a particular problem. For example, a software for payroll of an organization, attendance system for student, software for ticket reservation etc.
- **Packaged Software:** Packaged software is readymade, bug free, advance and standard software in special works. All word processing, database management, graphics, animation software are packaged software. These software are all equipped with all essential tools to enhance user productivity.
- **Utility Software:** These are special types of application software which helps us to fine tune the performance of a computer, prevent unwanted actions or perform system related tasks such as checking for virus and removing virus, sending fax, file copying, editors, system utilities which provide information about the current state of the use of files memory, users and peripherals, e.g. disk info, check disk, debuggers for removing "bugs" from programs.

1.6 Program

Basic commands that instruct the computer system to do something are called instructions. An organized list of instructions that, when executed, causes the computer to behave in a predetermined manner is a program. A program is like a recipe. It contains a list of ingredients (called variable) and a list of directions (called statements) that tell the computer what to do with the variables. The variables can represent numeric data, text or graphical images. Without programs, computer are useless.

1.7 Programming Language

- A programming language is a standardized communication technique for describing instructions for a computer.
- Each programming language has a set of syntactic and semantic rules used to define computer programs.
- A language enables a programmer to precisely specify what data computer is to act upon, how these data are to be stored/transmitted and what actions are to be taken under circumstances.
- Programming languages are another kinds of software which enables us to develop different kinds of software.
- Programming languages are classified mainly in two categories on the basis of creating instructions.
    - Low level languages
    - High level languages

1.7.1 Low level languages

- These are much closer to hardware. Before creating a program for a hardware, it is required to have through knowledge of that hardware.
- A program cannot be run on different hardware. Low level languages are specific to hardware. Low level language is also divided into two types:
    - Machine language
    - Assembly language

1.7.1.1 Machine language

- Machine level language are the lowest-level programming language. A computer understands programs written only in the machine language. While easily understood by computers, machine language are almost impossible for humans to use because they consist entirely of numbers.
- Ultimately, machine code consist entirely of the 0's and 1's of the binary number system. Which are also called bits.
- Early computers were programmed using machine language. Programs written in machine language are faster and efficient.
- Writing program in machine language is very tedious, time consuming, difficult to find bugs in longer programs.

1.7.1.2 Assembly language

- To overcome the difficulties of programming in machine language assembly language were developed.
- An assembly language contains the same instructions as a machine language, but each instruction and variable have a symbol instead of being just numbers. These symbols are called mnemonic. For example if 78 is the number to add two numbers, ADD could be used to replace it.
- After developing assembly language, it was easier to program using symbols instead of numbers. But the program written in assembly language must be converted to machine language which could be done by assembler.
- Each type of CPU (central processing unit) has its own machine language and assembly language. So an assembly language program written for one type of CPU won't run on another. This shows that assembly language is also machine dependent and time consuming. Programmers still use assembly language when speed is essential or when they need to perform an operation that isn't possible in a high-level language.

  - **Assembler:** An assembler is a program (software) which translates the program written in assembly language to machine language. Assembler produces one machine instruction for each source instructions and symbolic address to machine address. Assembler also includes the necessary linkage for closed subroutine, allocates area of storage, detects and indicates invalid source language instructions. The object program is stored on secondary storage media. The resulting program can only be executed when the assembly process is completed.

1.7.2 High level language

- The language are called high level language if their syntax is closer to human language.
- High-level language were developed to make programming easier. Most of the high level language are English like languages. They use familiar English words, special symbols (! & etc), and mathematical symbols (+,-,*,/ etc) in their syntax. Therefore high level language are easier to read, write, understand and programming. Which is the main advantage of the high level language over low level language.
- Each high level language has their own set of grammar and rules to represent set of instructions, programming language such as C, C++, JAVA, FORTRAN (acronym derived from the IBM Mathematical FORmula TRANslating System) or Pascal, Lisp, BASIC (Beginner's All-purpose Symbolic instruction Code) are some example of high level language.
- These enables a programmer to write programs that are more or less independent of a particular type of computer. Like assembly language programs, programs written in a high level language also need to be translated into machine language. This can be done either by a **compiler** or an **interpreter**.

1.8 Some terms

Source code: Initially, a programmer writes a program in a particular programming language like C, C++ and FORTRAN etc. This form of the program is called the source program, or more generically, source code. Source code is the only format that is readable by humans. The source code consist of instructions in a particular language. When we purchase programs we usually receive them in their machine-language format. This means that we can execute them directly, but we cannot read or modify them.

Object code: the code produced by a compiler is called object code. It is an intermediary form. Object code is often the same as or similar to a computer's machine language. Object code need to be converted into executable code using linkers.

Executable code: code that is ready to run is called executable code or machine code.

Compile: To transform a program written in a high level programming language (source code) into object code

Linker: Many programming language allows us to write different pieces of code, called modules, separately. This simplifies the programming task because we can break a large program into small, more manageable pieces. Eventually, we need to put all the modules together. This is the job of the linker. Therefore, a linker is a program that combines object modules to form an executable program. In addition to combining modules, a linker also replaces symbolic addresses with real addresses. We need to link a program even if it contains only one modules. A linker is also called a binder.

Link: In programming, the term link refers to execution of a linker.
Run: To execute a program.

1.9 Compiler

- A program that translates source code into object code. The compiler looks at the entire portion of source code and recognizes the instructions.
- Every high level programming language (except strictly interpretive language) comes with a compiler.
- A compiler defines the acceptable instructions. Because compilers translate source code into object code which is unique for each type of computer.
- Many compilers are available for the same language. For example we have been running Turbo C compiler in windows environment and gcc compiler in Linux system.

1.9.1 Compilation process

- The process of translation from high level language (source code) to low level language (object code) is called compilation. In this process, source code must go through several steps before it becomes an executable program.
  - The first step is to pass the source code through a compiler, which translates the high level language instructions into object code.
  - The final step in producing an executable program is to pass the object code through a linker. The linker combines modules and gives real values to all symbolic address, thereby producing machine code.
  - Compilation process ends by resulting an executable program.
  - The compiler stores the object and executable files in secondary storage.
  - If there is any illegal instructions in the source code, compiler lists all the errors during compilation.

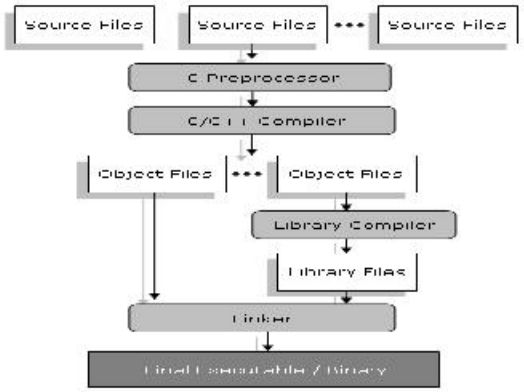


Figure 1-2: Illustration of Compilation Process

1.10 Interpreter

- An interpreter is a program which converts each high level program statement into machine code just before the program statement is to be executed. Translation and execution occur immediately one statement at a time. Interpreter directs the CPU to obey each program statement at a time.

1.11 Comparison between compiler and interpreter

Compiler	Interpreter
○ Compiler converts all the statement in source code to object code and finally in executable code resulting an exe file	○ Whereas interpreter converts each statement before executing it. It does not produce an exe file
○ Compiler require some time before producing an executable program	○ Interpreter can execute a program immediately
○ Once compiled program does not need to recompile for next run. Therefore, executable programs produces by compiler run much faster and efficient.	○ For running next time, it is required to repeat the process from the beginning. So it is slower process.
○ Immediate editing and executing a program is costly process because it take long time for compilation process if the program is long	○ It is possible to execute the edited program immediately. Therefore, it can be used in software developing phase and in learning phase for students.
○ C,C++, FORTRAN are example of compiled language	○ BASIC, LISP are interpreter language.

1.12 Firmware

Firmware is software that is embedded in a hardware device e.g. in a read-only memory (ROM) chip, erasable programmable read-only memory (EPROM) chip or as a binary image file that can be uploaded onto existing hardware by a user by using special external hardware. The BIOS in IBM-compatible personal computers is an example of firmware.

1.13 Traditional and structured Programming concepts

- In traditional programming, we start with a problem to solve. We figure out how to break the problem into smaller parts, then each part into smaller parts still. At each stage we think about how to do things. To do something means first doing one thing, then another, then yet another. So, we divide and conquer with an emphasis on doing.
- Structured programming (sometimes known as modular programming) is a subset of procedural programming that enforces a logical structure on the program being written to make it more efficient and easier to understand and modify.
- Structured programming frequently employs a top-down design model, in which developers map out the overall program structure into separate subsections. A defined function or set of similar functions is coded in a separate module or submodule, which means that code can be loaded into memory more efficiently and that modules can be reused in other programs. After a module has been tested individually, it is then integrated with other modules into the overall program structure.
- Structured programming was first suggested by Corrado Bohm and Guiseppe Jacopini. The two mathematicians demonstrated that any computer program can be written with just three structures: decisions, sequences, and loops. Edsger Dijkstra's subsequent article, goto Statement Considered Harmful was instrumental in the trend towards structured programming. The most common methodology employed was developed by Dijkstra. In this model (which is often considered to be synonymous with structured programming, although other models exist) the developer separates programs into subsections that each have only one point of access and one point of exit.
- Almost any language can use structured programming techniques to avoid common pitfalls of unstructured languages. Unstructured programming must rely upon the discipline of the developer to avoid structural problems, and as a consequence may result in poorly organized programs. Most modern procedural languages include features that encourage structured programming.

2 Programming Logic

2.1 Problem solving

- Problem is defined as the difference between an existing situation and a desired situation, that is, in accordance with calculation; a problem is numerical situation and has complex form. Solution is desired situation and has simplest form. If a problem is solved by computing using machine called computer, then such process is called Problem Solving using Computer.
- It is important to give a clear, concise problem statement. It is also called problem definition. The problem definition should clearly specify the following tasks.

Objective: The problem should be stated clearly so that there will not be the chance of having right solution to the wrong problem. Simple program can be stated easily but for complex problem may need a complex analysis with careful coordination of people, procedures and programs.
Output requirements: We must know what exactly we are expecting out of the system.
Input requirements: To get the desired output, it is required to define the input data and source of input data.
Processing requirements: It is required to clearly define processing requirements to convert the given input data to the required output. In processing requirements, there may be hardware platform, software platform, manpower etc.
Evaluating Feasibility: It is one of the important phases where we mainly decide whether the proposed software development task is technically and economically feasible.

2.2 Algorithms

- An algorithm is defined as a set or ordered steps or procedures necessary to solve a problem.
- It is s step wise presentation of procedures of program in simple English.
- To be an algorithm, the set of steps must be unambiguous and have a clear stopping point. Each step tells what task is to be performed.

Example 2-1:

Problem: There are 50 students in a class who appeared in their final examination. Their mark sheets have been given to you. Write an algorithm to calculate and print the total number of students who passed in first division.

Algorithm:

Step 1: Start
Step 2: Initialize Total First Division and Total Mark sheet checked to zero i.e.
total_first_div = 0;
total_marksheet_chkd = 0;
Step 3: Take the mark sheet of the next student.
Step 4: Check the division column of the mark sheet to see if it is I: if no, go to step 6.
Step 5: Add 1 to Total First Division i.e.
total_first_div +1;
Step 6: Add 1 to Total Mark sheets checked i.e.
total_marksheet_chkd +1;
Step 7: Is Total Mark sheets checked = 50: if no go to step 3
Step 8: Print Total First Division.
Step 9: Stop (End)

❖ An excellent algorithm should have the following properties

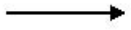
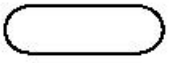
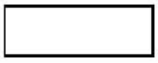
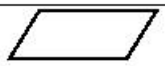
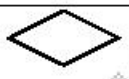
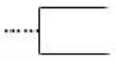
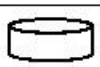
Finiteness: Each algorithm should have finite number of steps to solve a problem.
Definiteness: The action of each step should be defined clearly without any ambiguity.
Inputs: Inputs of the algorithms should be defined precisely, which can be given initially or while the algorithm runs.
Outputs: Each algorithm must result in one or more outputs.
Effectiveness: It should be more effective among the different ways of solving the problem.

2.3 Flowchart

- A flowchart is a pictorial representation of an algorithm that uses boxes of different shapes to denote different types of instructions. The actual instructions are written within these boxes using clear and concise statements. These boxes are connected by solid lines having arrow marks to indicate the flow of operation, that is, the exact sequence in which the instructions are to be executed.
- Normally, an algorithm is first represented in the form of a flowchart and the flowchart is then expressed in some programming language to prepare a computer program. The main advantage of this two steps approach in program writing is that while drawing a flowchart one is not concerned with the details of the elements of programming language. Since a flowchart shows the flow of operations in pictorial form, any error in the logic of the procedure can be detected more easily than in the case of a program. Once the flowchart is ready, the programmer can forget about the logic and can concentrate only on coding the operations in each box of the flowchart in terms of the statements of the programming language. This will normally ensure an error-free program.
- A flowchart, therefore, is a picture of the logic to be included in the computer program. It is simply a method of assisting the program to lay out, in a visual, two dimensional format, ideas on how to organize a sequence of steps necessary to solve a problem by a computer. It is basically the plan to be followed when a program is written. It acts like a road map for a programmer and guides him/her how to go from starting point to the final point while writing a computer program.
- Experienced programmers sometimes write programs without drawing the flowchart. However, for a beginner it is recommended that a flowchart be drawn first in order to reduce the number of errors and omissions in the program. It is a good practice to have a flowchart along with a computer program because a flowchart is very helpful during the testing of the program as well as while incorporating further modifications in the program.

❖ Flowchart Symbols

A flowchart uses boxes of different shapes to denote different types of instructions. The communication of program logic through flowcharts is made easier through the use of symbols that have standardized meanings. For example, a diamond always means a decision. Only a few symbols are needed to indicate the necessary operations in a flowchart. These symbols are standardized by the American National Standard Institute (ANSI). These symbols are listed below:

Symbol	Purpose	Description
	Flow line	Used to connect symbols and indicates the flow of logic
	Terminal (start/stop)	Used to indicate the start and end of a flowchart. One flow line exits from start and one enters to stop
	Processing	Used whenever data is being manipulated, most often with arithmetic operations. A single flow line enters and a single flow line exits.
	Input/Output	Used whenever information is entered into the flowchart or displayed from the flowchart. A single flow line enters and a single flow line exits.
	Decision	Used to represent operation in which there are two possible alternatives i.e. decision making and branching. One flow line enters and two flow lines (labeled true and false) can exits.
	Predefined process/ Function	Used to represent a function call, or variables are sent to another function to return data or an answer. A single flow line enters and a single flow line exits.
	On-page Connector	Used to connect remote flowchart portions on the same page. One flow line enters or exits.
	Off-page connector	Used to connect remote flowchart portions on different pages. One follow line enters or exits.
	comment	Used to add comments or clarification.
	Magnetic disk storage	Used to show the storage in magnetic disk. More than one flow line can enter and exit.

❖ Advantages of using Flowcharts

Communication: Flowchart are better way of communicating the logic of a system to all concerned.
Effective analysis: With the help of flowchart, problem can be analyzed in more effective way.
Proper documentation: Program flowcharts serve as a good program documentation, which is needed for various purposes.
Efficient Coding: The flowcharts act as a guide or blueprint during the system analysis and program development phase.
Proper Debugging: The flowchart helps in debugging process.
Efficient Program Maintenance: The maintenance of operating program becomes easy with the help of flowchart.

❖ Limitation of using flowchart

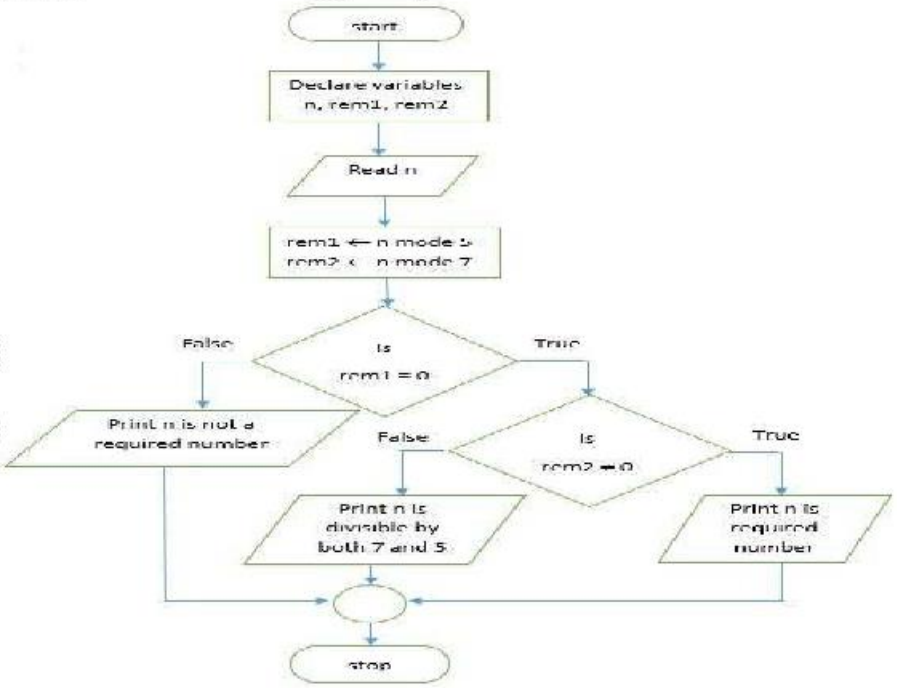
Complex Logic: Sometimes, the program logic is quite complicated, In that case, flowchart becomes complex and clumsy
Alternations and Modifications: If alternations are required the flowchart may require re-drawing completely.

Example 2-2: Write algorithm and draw a flowchart to check whether a number is exactly divisible by 5 but not by 7.

Algorithm

Step 1: Start
Step 2: Declare variable n, rem1, rem2
Step 3: Read n
Step 4: Find remainders
rem1 ← n mode 5
rem2 ← n mode 7
Step 5: if rem1 = 0
if rem2 ≠ 0
print n is required number
else
print n is divisible by both 5 and 7
else
print n is not required number
Step 6: Stop

Flowchart:



2.4 Coding

In order to make a program in any programming language, what you have written is known as code. The act of writing code in a computer language is known as coding. In other words, code is a set of instruction that a computer can understand.

2.5 Compilation and execution

- Program written in high level language need to be converted to low level. Compiler does this task
- The process by which source codes of a computer (programming language are translated into machine codes is known as compilation. After compilation if everything is ok, the code is going under other process that is known as execution. We can get the required output after execution process.

2.6 Debugging and testing

Debugging and testing is the process of detecting and removing errors in a program, so that the program produces the desired results on all occasions.

2.6.1 Types of errors

Generally errors are classified into following types:

- **Syntax Error:** Any violation of rules of the language results in syntax error.
- **Run-time Error:** Errors such as mismatch of data types or referencing an out of range array element go undetected by the compiler. A program with these mistakes will run but produce the erroneous result.
- **Logical Error:** These errors are related to the logic of the program execution. Such action as taking the wrong path, failure to consider a particular condition and incorrect order of evaluation of statements belong to this category.
- **Latent Error:** These are hidden errors that shows up only when a particular set of data is used.

  E.g.-

  ```
  r = (x + y) / (p - q)
  ```

  This expression generates error when p=q

2.6.2 Debugging

Debugging is the process of isolating and correcting errors. Different type of debugging techniques are as follows:

- **Error Isolation:**
  Error isolation is used for locating an error resulting in a diagnostic message. If we do not know the general location of the error, we can generally find the location of the error by temporarily deleting a certain portion of program and rerunning the program to see whether the error is again appeared or not. If the error is not appeared again, we know that the error is in the deleted code. If the error again appears, we know that the deleted portion is error free. Temporary deletion is accomplished by surrounding the instructions with comment markers (/* */). Similarly, we can put different unique printf statement in different parts (blocks) of the program to check whether that parts (blocks) of the program is executed or not.
- **Tracing:**
  In this technique, printf statement is used to print the value of some important variables at different stages of program. By observing these values, we can decide that whether the assigned values are correct or not. If any value is incorrect, we can easily fix the location of the error. Similarly we can display and observe the value of variables that are calculated internally and assigned to the variables.
- **Watch Values:**
  A watch value is the value of a variable or an expression, which is displayed continuously as the program executes. Thus, we can see the changes in a watch value as they occur, in response to the program logic. By inspecting these values carefully, we can determine where the program begins to generate an incorrect or unexpected value. In Turbo C++ IDE, to define the watch values, go through the following steps

  Debug->Watchs->Add watch…->specify variables or expression to watch its value->OK.

- **Breakpoints:**
  A break point is a temporarily stopping point within a program. Each breakpoint is associated with particular instruction within the program. When the program is executed, the program execution will temporarily stop at the breakpoint before the instruction is executed. The execution may be resumed until the next breakpoint is encountered. Breakpoints are often used in conjunction with watch values, by observing the current watch value at each break points as the program executes. In Turbo C++, go through the following steps to set breakpoints

  Debug->Add->Breakpoint->provide the requested information in the dialog box.

  Or

  Select a particular line within the program->press function key F5.

  To disable the breakpoint again press function key F5.

- **Stepping:**
  The process of executing one instruction at a time is called stepping. We can determine which instruction produce erroneous result or generate error message by stepping through the entire program. Stepping is often used with watch values, allowing us to trace the entire history of a program as it executes. Thus, we can observe changes to watch value as they happen. This allows us to determine which instructions generate erroneous results.
  In Turbo C++, **stepping can be done by pressing function key F7.**

2.6.3 Testing

- Software testing is the process of executing a program or system with the intent of finding errors. It involves any activity aimed at evaluating an attribute or capability of a program or system and determining that it meets its required result.
- Testing is usually performed for the following purposes:
  To improve quality.
  For verification and validation
  For reliability estimation.
- Testing process may include the following two stages:

- **Human Testing:**
  Human testing is an effective error detecting process and is done before the computer based testing begins. Human testing method include code inspection by the programmer and test group and review by a peer group. The test is carried out statement by statement and is analyzed with respect to a checklist of common programming errors. In addition to finding the errors, the programming style and choice of algorithm are also reviewed.
- **Computer Based Testing:**
  Computer based testing involves two stages namely compiler testing and run time testing. Different type of debugging techniques are used to find the syntax error. If there is no syntax error, we need to find the logical and runtime errors. Which can be known after running the program. Run time error may produce the run time error message such as null pointer assignment and stack overflow. After removing these errors, it is required to run the program with test data to check whether the program is producing the correct result or not. Program testing can be done either at module (function) level or at program level. The module level test is the unit test. Where all the units are tested. They should be integrated to perform the desired functions. An integration testing is done to find the errors associated with interfacing.

2.7 Program Documentation

- Program Documentation refers to the details that describe a program. Some details may be built-in as an integral part of the program. These are known as internal documentation. Two important aspects of internal documentation are; selection of meaningful variable names and the use of comments. Selection of meaningful names is crucial for understanding the program. For example,

  Area = Breadth * Length; is more meaningful than A = B * L

  And comments are used to describe actions parts and identification in a program.

- Program documentation starts from the starting of the software development life cycles. It keeps most of the information of all phases while developing projects, in design phase we need to develop the algorithm and flowchart of the software being developed and these material should be documented properly for future reference. Similarly, there is particular output in each phase for documentation. Documentation is used for future reference for both the original programmer and beginner. The final document should contain the following information:

- A program analysis document with objectives, inputs, outputs and processing procedures.
- Program design document algorithm and detailed flowchart and other appropriate diagrams.
- Program verification documents, with details of checking, testing and correction procedures along with the list of test data.
- Log is used to document future program revision and maintenance activity.

3 Variables and data types

3.1 Introduction to C

C is a general-purpose, structured programming language. Its instructions consists of terms that resemble algebraic expression, augmented by certain English keywords such as if, else, for, do and while, etc. C contains additional features that allow it to be used at a lower level, thus bridging the gap between machine language and the more conventional high level language. This flexibility allows C to be used for system programming (e.g. for writing operating systems as well as for applications programming such as for writing a program to solve mathematical equation or for writing a program to bill customers). It also resembles other high level structure programming language such as Pascal and FORTRAN.

3.2 Historical Development of C:

C was an offspring of the 'Basic Combined Programming Language' (BCPL) called B, developed in 1960s at Cambridge University. B language was modified by Dennis Ritchie and was implemented at Bell Laboratories in 1972. The new language was named C. Since it was developed along with the UNIX operating system, it is strongly associated with UNIX. This operating system was developed at Bell Laboratories and was coded almost entirely in C.

C was used mainly in academic environments for many years, but eventually with the release of C compiler for commercial use and the increasing popularity of UNIX, it began to gain populuidespread support among compiler professionals. Today, C is running under a number of operating systems including Ms-DOA. C was now standardized by American National Standard Institute. Such type of C was named ANSI C.

3.3 Importance of C:

Now-a-days, the popularity of C is increasing probably due to its many desirable qualities. It is a robust language whose rich set of built-in functions and operators can be used of built-in functions and operators can be used to write any complex program. The C compiler combines the capabilities of an assemble language with the features of a high-level language and therefore it well suited for writing both system software and business packages. In fact, many of the C compilers available in the market are written in C.

Programs written in C are efficient and fast. This is due to its variety of data types and powerful operators. It is many times faster than BASIC (Beginners All Purpose Symbolic Instruction Code – a high level programming language).

There are only 32 keywords and its strength lies in its built-in functions. Several standard functions are available which can be used for developing programs. C is highly portable. This means that C programs written for one computer can be seen on another with little or no modification. Portability is important if we plan to use a new computer with a different operating system.

C Language is well suited for structure programming thus requiring the user to think of a problem in terms of function modules or blocks. A proper collection of these modules would make a complete program. This modular structure makes program debugging, testing and maintenance.

Another important feature of C is its ability to extend itself. A C program is basically a collection of functions that are supported by the C library. We can continuously add our own function to the C library. With the availability of a large number of functions, the programming task becomes simple.

3.4 Basic Structure of C programs:

Every C program consists one or more modules called function. One of the function must be called main( ).A function is a sub-routine that may include one or more statements designed to perform a specific task. A C program may contain one or more sections shown in figure 3-1:

The documentation section consists of a set of comment lines giving the name of the program, the author and other details which the programmer would like to use later. The link section provides instructions to the compiler to link function from the system library. The definition defines all the symbolic constants. There are some variables that are used in more than one function. Such variables are called global variables and are declared in global declaration section that is outside of all the function.

Every C program must have one main( ) function section. This section consists two parts: declaration part and executable part. The declaration part declares all the variables used in the executable part. These two parts must appear between the opening and the closing braces. The program execution begins at the opening braces and ends at the closing brace. The closing brace

of the main ( ) function section is the logical end of the program. All the statements in the declaration and executable parts ends with a semicolon.

The subprogram section contains all the user-defined functions that are called in the main ( ) function. User-defined functions are generally placed immediately after the main ( ) function, although they may appear in any order. All section, except the main ( ) function section may be absent when they are not required.

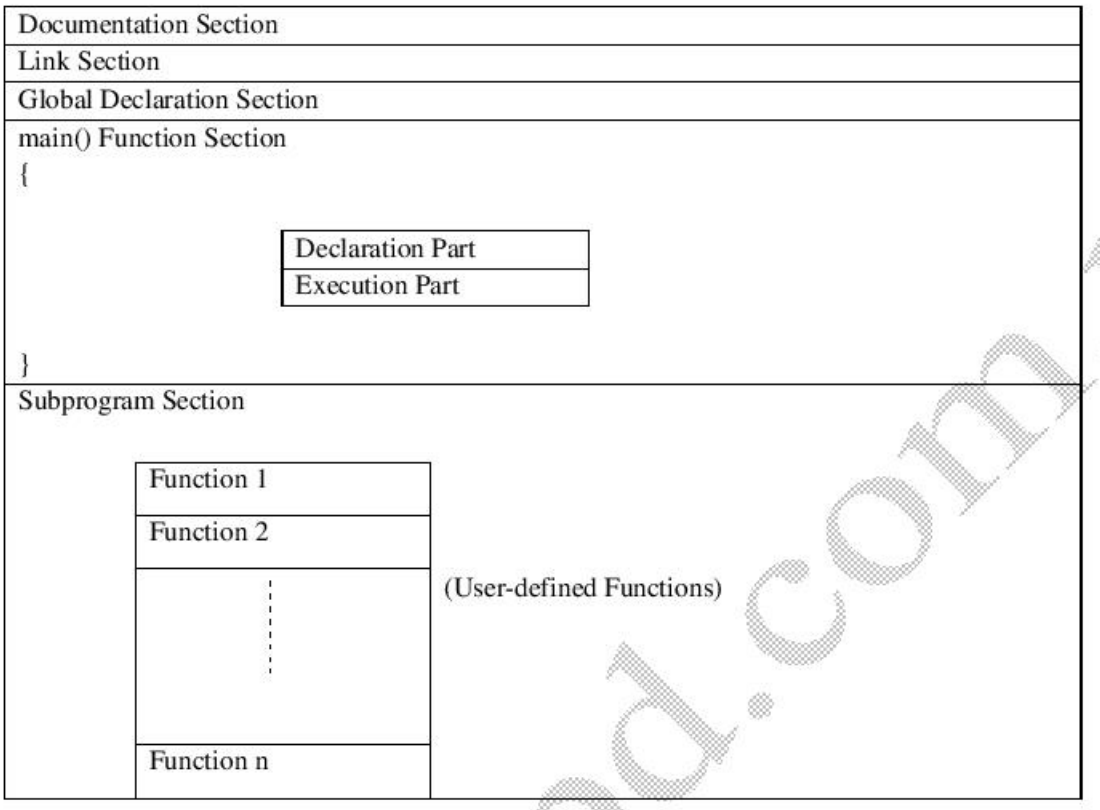


Figure 3-1: Basic Structure of a C program

Example 3-1: C Basic Program

```
/* This is the first C program. It prints Hello World on the screen*/

#include<stdio.h>
#include<conio.h>
int main()
{
        printf("Hello World");
        getch();
        return 0;
}
```

Output:

Hello World

3.5 Character Set

- C uses the uppercase letters A to Z, the lowercase letters a to z, the digits 0 to 9, and certain special characters as building blocks to form basic program elements (e.g. constants, variables, operators, expressions, etc).
- The characters available in C are categorized into following types
  1) Letter -> Uppercase- A-Z, Lowercase- a-z
  2) Digits -> 0….9
  3) Special Characters -> , : ; { } [ ] ( ) % & $
  4) White space -> horizontal tab, blank space, new line, carriage return, form feed

❖ Special characters:

;	comma	%	Percent sign
.	period	&	ampersand
:	colon	^	caret
;	semicolon	*	asterisk
?	Question mark	<	Opening angle bracket(less than)
‘	apostrophe	>	Closing angle bracket (greater than)
“	Quotation mark	(	Left parenthesis
!	Explanation mark	)	Right parenthesis
/	slash	[	Left bracket
\	Back slash	]	Right bracket
~	Tilde	{	Left brace
_	Under score	}	Right brace
$	Dollar sign	#	Number sign (Hash)

3.6 Escape Sequence

- Certain nonprinting character, as well as the backslash (\) and apostrophe (`), can be expressed in terms of escape sequences. An escape sequence always begins with a backslash and is followed by one or more special characters. For example, a linefeed (LF), which is referred to as a newline in C, can be represented as \n. Such escape sequences always represent single characters, even though they are written in terms of two or more characters.
- The commonly used escape sequences are listed below:

Character	Escape Sequence	ASCII Value
bell (alest)	\a	007
backspace	\b	008
horizontal tab	\t	009
vertical tab	\v	011
newline (line feed)	\n	010
form feed	\f	012
carriage return	\r	013
quotation mark (”)	\”	034
apostrophe (`)	\`	039
Question mark (?)	\?	063
backslash (\)	\\	092
null	\0	000

- Several character constants are expressed in terms of escape sequences are

  ‘\n’ ‘\t’ ‘\b’ ‘\’’ ‘\\’ ‘\’’’

  The last three escape sequences represent an apostrophe, backslash and a quotation mark respectively.
  Escape Sequence ‘\0’ represents the null character (ASCII 000), which is used to indicate the end of a string. The null character constant ‘\0’ is not equivalent to the character constant ‘0’.
  The general form ‘\000’ represents an octal digit (0 through 7). The general form of a hexadecimal escape sequence is \xhh, where each h represents a hexadecimal digit (0 through 9 and a through f).

3.7 C-TOKENS

In a passage of text, individual words and punctuation marks are called tokens. Similarly, in a C program the smallest individual units are also known as C tokens. C has following types of tokens:

- Keyword
- Identifier
- Constant
- String
- Special Symbols
- Operators

3.8 Keywords

- There are certain reserved words, called keywords that have standard, predefined meanings in C. These keywords can be used only for their intended purpose, they cannot be used as programmer-defined identifiers.
- Note all keywords must be written in lowercase.
- The standard keywords are

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

3.9 Identifier

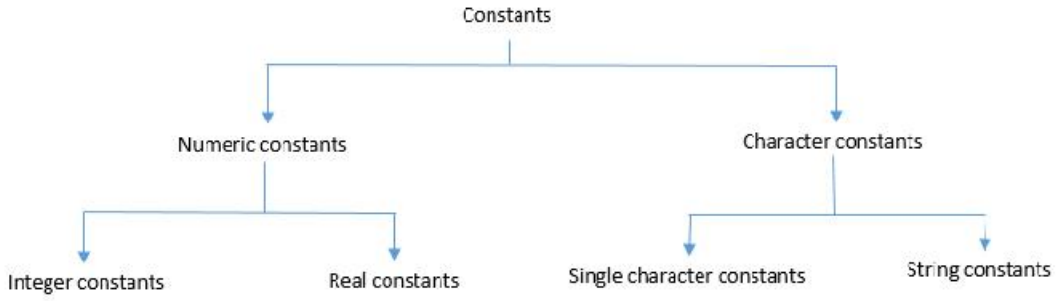
- Identifier are the names given by user to various program elements such as variables, function and arrays.

❖ Rules of identifier

1. Must contain of only letter, digits or underscore.
2. First character must be an alphabet, may also begin with an underscore, though this is rarely done in practice.
3. The length of identifier should not be greater than 31 characters.
4. Cannot use a keyword.
5. Underscore is permitted between two words but white space is not allowed.
6. It is case sensitive i.e. uppercase and lowercase letters are not interchangeable.

3.10 Constants

- Constants refer to fixed values that do not change during the execution of program.



3.10.1 Integer constants

- An integer constant is an integer valued number. Thus it consists of a sequence of digits.
- Integer (number) constants can be written in three different number systems: decimal (base 10), octal (base 8) and hexadecimal (base 16).
- A **decimal integer constant** can consists of any combination of digits taken from the set 0 through 9. If the constant contains two or more digits, the first digit must be something other than 0. Several valid decimal integer constants are shown below:

  0 1 143 5280 12345 9999

  The following decimal integer constants are written incorrectly for reason stated:

12,452	illegal character (,)
36.0	illegal character (.)
10 20 30	illegal character (blank space)
123_45_6743	illegal character (-)
009	the first digit cannot be zero.

- An **octal integer constant** can consist of any combination of digits taken from the set 0 through 7. However, the first digit must be 0, in order to identify the constant as an octal number.

  Valid octal number (integer) constants are shown below:

  0 01 0743 07777

  The following octal integer constants are written incorrectly for the reason stated:

743	does not begin with 0.
05280	illegal character (8)
777.777	illegal character (.)

- A **hexadecimal integer constant** must begin with either 0_x or 0X. It can then be followed by any combination of digits taken from the sets 0 through 9 and a through f (either upper or lower case). The letters a through f (or A through F) represent the (decimal) quantities 10 through 15 respectively.

  Several valid hexadecimal integer constants are shown below:

  0_x 0X1 0X7FFF 0xabcd

  The following hexadecimal integer constants are written incorrectly for the reason stated:

0X12.34	illegal character (.)
013E38	doesn't begin with 0_x or 0X.
0_x.4bff	illegal character (.)
0XDEFG	illegal character(G)

- **Unsigned and Long Integer Constants:**

  Unsigned integer constants may exceed the magnitude of ordinary integer constants by approximately a factor of l, though they may not be negative. An unsigned integer constant can be identified by appending the letter U (either upper or lowercase) to the end of the constant.

  Long integer constants may exceed the magnitude of ordinary integer constants, but require more memory within the computer. A long integer constant can be identified by appending the letter L (either upper or lowercase) to the end of the constant.

  An unsigned long integer may be specified by appending the letters UL to the end of the constant. The letters may be written in either upper or lowercase. However, the U must precede the L.

  Several unsigned and long integer constants are shown below:

Constant	**Number System**
50000 U	decimal (unsigned)
123456789 L	decimal (long)
123456789 UL	decimal (unsigned long)
0123456 L	octal (long)
0777777 U	octal (unsigned)
0X50000 U	hexadecimal (unsigned)
0XFFFFFUL	hexadecimal (unsigned long)

3.10.2 Real Constants/ Floating point Constants

- A floating point constant is a base 10 number that contains either a decimal point or an exponent (or both).
- Floating point constants have a much greater range than integer constants.
- Typically, the magnitude of a floating point constant might range from a minimum value of approximately 3.4E-38 to a maximum of 3.4E+38.

  Several valid floating point constants

0.	1.	0.2	827.602
500.	0.000743		12.3
2E.8	0.006e.3		1.6667e+8

The following are not valid floating point constants for the reason stated.

1	Either a decimal point or an exponent must be present.
1,000.0	Illegal character (,)
2E+10.2	The exponent must be an integer (it cannot contain a decimal point)
3E 10	Illegal character (blank space) in the exponent.

3.10.3 Character Constants

- A character constant is a single character enclosed within a pair of single quotation marks.
- Character constants have integer value known as ASCII (American Standard Code for Information Interchange)
- Several character constant and their corresponding values, as defined by ASCII character set are shown below:

Constant	Value
'A'	65
'x'	120
'3'	51
'?'	63
' '	32

3.10.4 String Constants

- A string constant is a sequence of characters enclosed in double quotes. The characters may be letters, numbers, special character and blank spaces.
- E.g.- "ADVANCED" "A" "green" "Hello" "302"
- The compiler automatically places a null character (\0) at the end of every string constant, as the last character within the string (before the closing double quotation mark). This character is not visible when the string is displayed.
- A character constant (e.g. 'A') and the corresponding single-character string constant ("A") are not equivalent. A character constant has an equivalent integer value, whereas a single character string constant does not have an equivalent integer value and in fact, consists of two characters – the specified character followed by the null character (\0).

3.10.5 Symbolic Constants

- A symbolic constant is simply an identifier used in place of constant.
- They are usually defined at the start of a program, so that is changes are required they can be easily located.
- E.g.

```
#define PI 3.1415
```

Where #define is a preprocessor directive

3.11 Variables

- A variable is an identifier that is used to represent some specified type of information within a designated portion of the program.
- Simply variables are the name given to store region of memory.
- The value of variable changes during the execution of the program.
- The memory allocation is done at the time of variable declaration.
- A variable name can be chosen by the programmer in a meaningful way to reflect the nature of program.

❖ Array Variables

An array is an identifier that refers to a collection of data items that have the same name. The data item must be of the same type. The individual data item are represented by their corresponding array element i.e. first data item is represented by the first array element.

❖ Declarations

A declaration associates a group of variables with a specific data type. All variables must be declared before they can appear in executable statements.

A declaration consists of a data type, followed by one or more variable names, ending with a semicolon. Each array variable must be followed by a pair of square brackets, containing a positive integer which specifies the size (i.e. the number of elements) of the array.

A C program contains the following type declarations:

Syntax: **Data-type variable-name;**

Example:

```
int a,b,c;
float  root1,root2;
char flag,text [80],
```

Thus, a, b and c are declared to be integer variables, root1 and root 2 are floating variables, flag is a char-type variable and text is an 80-element, char-type array. Square brackets enclosing the size specification for text.

These declarations could also have been written as follows:

```
int    a;
int    b;
int    c;
float  root1;
float  root2;
char   flag;
char   text[80];
```

❖ Assigning Values to Variables

Values can be assigned to variables using assignment operator "=".

Syntax: **variable-name = value;**

Example:

```
int a;
a = 10;
```

It is also possible to assign a value to a variable at the time the variable is declared. The process of giving initial value to the variable is called initialization of the variable

Example:

```
int a = 10,b=5;
float x=10.5;
```

The data item can be accessed in the program simply by referring to the variable name. Data type associated with the variable cannot be changed. However variables hold the most recently assigned data.

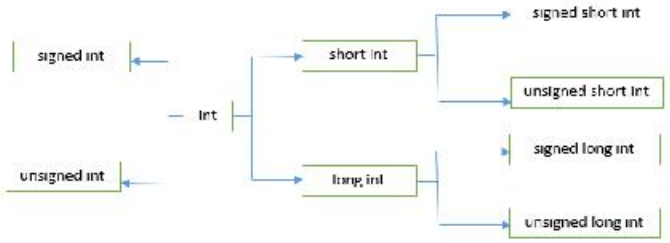
3.12 Data types

- The data type specifies the range of values that a variable or constant can hold and how that information is stored in the computer memory
- C language is rich in its data types
- The variety of data types available allows the programmer to select the type appropriate to the needs application as well as machine.
- There are three cases of data types:

1. Basic data types (Primary or Fundamental)
   - char
   - int
   - float
   - double
2. Derived data types
   - function
   - pointer
   - array
3. User defined data types
   - structure
   - union
   - enumeration

3.12.1 Fundamental Data Types

1. **char:** the char type is used to represent individual characters so, it generally requires only one byte of memory. Each char type has an equivalent integer interpretation and hence a char is regarded as a special kind of short int.
   [1byte = 8bit]
2. **int:** it represents an integer quantity and its typical memory requirement is 2 bytes or one word [2byte = 16 bit]



3. **float:** it represents a floating point number i.e. a number containing a decimal point or/and an exponent. Its memory requirement is generally 4 bytes.
4. **double:** it represents a double-precision floating point number i.e. a more significant figure and an exponent which may be larger in magnitude.
5. **void type:** The **void** is used to indicate that an expression has no value. No variable can be declared with such a type, but expression may be cast to void. For example:
   (void)printf("Nepal");
   Specifically indicates to the compiler that the returns value from printf (an integer) is to be ignored.
   As such, the statement
   a = (void)printf("Nepal");
   is illegal because the assignment operator expects a value to be returned for assignment.

❖ **Size and range of data types on 16-bit machine**

Data type	keyword	Size (in byte)	Range
Character or signed character	char or signed char	1	-128 to 127
Unsigned character	Unsigned char	1	0 to 255
Integer or signed integer	int or signed int	2	-32768 to 32767
Unsigned integer	Unsigned int	2	0 to 65535
Short integer or signed short integer	Short int or signed short int	1	-128 to 127
Unsigned short integer	Unsigned short int	1	0 to 255
Long integer or signed long integer	Long int or signed long int	4	-2147483648 to 2147483647
Unsigned long integer	Unsigned long int	4	0 to 4294967295
Floating point	float	4	3.4e-38 to 3.4e+38
Double precision floating point	double	8	1.7e-308 to 1.7e+308
Extended double precision floating point	Long double	10	3.4e-4932 to 1.1e+4932

3.13 Qualifiers

- Qualifier modify the behavior of the variable to which they are applied

  E.g.-

  integer qualifier can be defined as short int, long int or unsigned int.

  Here short, long, signed and unsigned are type qualifier.
- There are 4 types of qualifiers

Size qualifier: It alter the size of the basic data types. The keyword long and short are two size qualifiers. Both can be applied to int only, long can be applied to double.

Sign qualifier: It specify whether a variable can hold both negative and positive numbers, or only positive numbers. The keyword signed and unsigned are two sign qualifier. These type of qualifiers can be applied to the data types int and char only.

const: An object declared to be const cannot be modified by a program.

e.g.- **const int r=20;**

Volatile: A variable should be declared volatile whether its value can be changed by some external source from outside the program.

e.g.- **volatile int p = 10;**

3.14 Enumeration

- Enumeration is a technique that defines a set of integer type constants,
- In C, an enumeration is of type int. Enumeration is useful to define a set of constants instead of using multiple #define.
- Enumeration is a way of creating of user define data types.
- It starts with 0(zero) by default and value is incremented by 1 for the sequential identifiers in the list.

  Syntax: **enum identifier{value 1,value 2,…..value n};**
- Enum example in C
  - enum month {Jan, Feb, Mar}; /* Jan, Feb and Mar variables will be assigned to 0, 1 and 2 respectively by default*/
  - enum month {Jan=1, Feb, Mar}; /* Feb and Mar variables will be assigned to 2 and 3 respectively by default*/
  - enum month {Jan=20, Feb, Mar}; /* Jan is assigned to 20, Feb and Mar variables will be assigned to 21 and 22 respectively by default*/

Example 3-2:

```
#include<stdio.h>
#include<conio.h>
int main()
{
        int i;
        enum week{sun,mon,tue,wed,thur,fri,sat};
        for(i=sun;i<=sat;i++)
        {
                printf("%d\t",i);
        }
        getch();
        return 0;
}
```

Output:

```
0    1    2    3    4    5    6
```

3.15 Typedef Statement:

C supports a feature known as "type definition" that allows users to define an identifier that would represent an existing data type. The user-defined data type identifier can later be used to declare variables. It takes the general form:

typedef type identifier;

where type refers to an existing data type and identifier refers to the "new" name given to the data type. The existing data type may belong to any class of type, including the user defined ones. The new type is new only in name, but not the data type. **typedef** cannot create a new type. Some examples of type definition are:

typedef int units;
typedef float marks;

Here, units represent int and marks represents float. They can be later used to declare variables as follows:

units batch1, batch2;
marks subject1[50]; subjecct2[50];

batch1 and **batch2** are declared as int variable and **subject1[50]** and **subject2[50]** are declared as 50 element floating point array variables. The main advantage of **typedef** is that we can create meaningful data type names for increasing the readability of the program.

3.16 Simple Input/Output Function

- In C language input and output functions are accomplished through library function. The header file for I/O functions is **<stdio.h>**. In C there are two types of I/O functions. They are console I/O and file I/O. Console I/O functions takes input from keyboard and produces output on the screen. The console I/O functions are also called Standard input/output functions.
- The console I/O functions are classified as shown below:
  - Formatted function
    - scanf
    - printf
  - Unformatted function
    - getchar()
    - putchar()
    - getch()
    - putch()
    - gets
    - puts

3.16.1 Formatted I/O Functions

- Formatted I/O means reading and writing data in formats which are desired by the user. The function used to input action is **scanf()** and output action is **printf().**

☐ printf() function

- **printf()** is used to print or display data on the console in formatted form.
- The formate of **printf()** is **printf("control string", list of arguments);**
- Control string contains the formatting instructions within quotes. This can contain
  - Character that are simply printed as they are
  - Conversion specifications with begins with format specifier (%) sign.
  - Escape sequences
- Arguments values get substituted in the place of appropriate format specifications. Arguments can be variable, constants, arrays or complex expressions.
- The percentage (%) followed by conversion character is called formate specifier. This indicates the type of the corresponding data item.

Format specifier	Meaning
% d or % i	decimal integers
% u	unsigned decimal integer
% x	unsigned hexadecimal (lower case letter)
% X	unsigned hexadecimal (upper case letter)
% o	octal
% c	character
% f	floating point
% s	strings
% lf	double
% ld	long signed integer
% lu	long unsigned integer
% p	displays pointer
% %	prints a % sign
% e	scientific notation (e lower case)
% E	scientific notation (e upper case)

☐ Minimum Field Width Specifier

- An integer placed between the % sign and format code is called minimum field width specifier or % number format code.
- E.g.- %5d.
- If string or number to be printed is longer than that minimum it will printed as such Minimum field width specifiers are used to make the O/P such that it reaches the minimum length specified.
- The following example demonstrates the working of minimum field width specifier

```
Double count;
Count = 10.51324;    // here total digits are 8, includes dot ( . )
operators.

printf (" %f", count);
printf (" %10f", count);
printf (" %010f", count);

Output
 10.51324
 10.51324      // inserts 2 blank spaces. So that total length = 10
digits
 0010.51324          // inserts 2 zeros. So that total length = 10 digits
```

☐ scanf() function

- The **scanf()** reads the input data from standard input device i.e. keyboard.
- The general format of the **scanf()** function is **scanf("format string" list of argument);**
- Where format string consist of format specifiers and arguments consist of address of variables. To this corresponding address the data read from keyboard is sent. The address of the variable is denoted by ampersand symbol '&'.
- Note that the values that are supplied through keyboard must be separated by either blank tabs or newlines. Escape sequences are not included in **scanf()** function.

```
(i) To read integer data:
      int i ;
      - - -
      scanf("%d", &i);

(ii)To read floating point data:
      floatf;
      - - -
      scanf("%f", &f);

(iii)  To read character data;
      char sam, john;
      - - -
      scanf("%c", &sam, &john);

(vii)  To read more than one data types at a time
      int i;
      float b;
      char c;
      string s;
      - - -
      scanf("%d %f %c %s", &i, &b, &c, &s );
```

3.16.2 Unformatted I/O Function

- A simple reading of data from keyboard and writing to I/O device, without any format is called unformatted I/O functions. This is classified as string I/O and character I/O.

❖ Character Input/Output (I/O)

- In order to read and output a single character character I/O functions are used. The functions under under character I/O are

  getchar() putchar() getch() getche()

☐ getchar() function

- single characters can be entered into the computer using the C library function **getchar()**. The getchar function is a part of the standard C I/O library. It returns a single character from a standard input device typically a keyboard.
- The function does not require any arguments, though a pair of empty parentheses must follow the word **getchar()**.
- The general syntax

  Character variable= getchar();

  Where character variable refers to some previously declared character variable.

E.g.-

```
char c;
c=getchar();
```

- Drawback with **getchar()** is that buffers the input until 'ENTER' key is pressed. This means that getchar does not see the characters until the user presses return. This is called line buffered input. This line buffering many leave one or more characters waiting in the input queue, which leads to 'errors' in interactive environment. Hence alternatives to **getchar()** is used.
- The two common alternative functions to **getchar()** are
  - **getch()**
    -this is a function gets a character from keyboard but does not echo to the screen. It allows a character to be entered without having to press the ENTER key afterwards
  - **getche()**
    -this is a function that gets a character from keyboard and echoes to screen. It allows a character to be entered without having press the ENTER key afterwards.

☐ putchar() function

- Single characters can be displayed using the C library function **putchar()**. This function is complementary to the character input function **getchar()**.
- The **putchar()** function, like **getchar()**, is a part of the standard C I/O library. It transmits a single character to a standard output device.
- The character being transmitted will normally be represented as a character type variable. It must be expresses as an argument to the function, enclosed in parentheses, following the word **putchar()**.
- The general syntax is

  putchar(character variable)

  where character variable refers to some previously declared character variable.

  E.g.-

```
char c = 'a';
putchar(c);
```

☐ String I/O

- In order to read and write string of characters the functions **gets()** and **puts()** are used **gets()** function reads the string and **puts()** function takes the string as argument and writes on the screen.
- E.g.-

```
char name[30];
puts("Enter your name");
gets(name);
puts("the name entered is");
puts(name);
```

3.17 Operators

- An operator is a symbol that tells the compiler to perform specific mathematical or logical manipulations.
- An operator is a symbol which acts on operand to produce certain result as output. The data items on which operators act upon are called 'operands'. For example in the expression **a + b; +** is an operator, **a** and **b** are operands.
- Based on the number of operands the operator acts upon, Operators can be classified as follows:
  - Unary operators: acts on a single operand. For example
    Unary minus(-5,-20, etc), address of operator(&a).
  - Binary operators: acts on two operands. For example +, -, %, /, *, etc.
  - Ternary operator: acts on three operands. For example
    conditional operator (?:)
- Based on the function, operators can be classified as following categories:
  - Arithmetic operator
  - Relational operator
  - Logical operator
  - Assignment operator
  - Increment and decrement operators
  - Bitwise operator
  - Conditional operator
  - Special operators

3.17.1 Arithmetic Operators

- C arithmetic operators are used to perform mathematical calculations like addition, subtraction, multiplication, division and modulus in C programs.
- 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
*	Multiplies both operands	A * B will give 200
/	Divides numerator by de-numerator	B / A will give 2
%	Modulus Operator and remainder of after an integer division	B % A will give 0
++	Increments operator increases integer value by one	++A will give 11
--	Decrements operator decreases integer value by one	--A will give 9

Example 3-3:

```
#include <stdio.h>
#include<conio.h>
int main()
{
        int a = 10;
        int b = 20;
        int c ;
        c = a + b;
        printf("Line 1 - Value of c is %d\n",c);
        c = a - b;
        printf("Line 2 - Value of c is %d\n",c);
        c = a * b;
        printf("Line 3 - Value of c is %d\n",c);
        c = a / b;
        printf("Line 4 - Value of c is %d\n",c);
        c = a % b;
        printf("Line 5 - Value of c is %d\n",c);
        c = ++a;
        printf("Line 6 - Value of c is %d\n",c);
        c = --a;
        printf("Line 7 - Value of c is %d\n",c);
        getch();
        return 0;
}
```

Output:

```
Line 1 - Value of c is 30
Line 2 - Value of c is -10
Line 3 - Value of c is 200
Line 4 - Value of c is 0
Line 5 - Value of c is 10
Line 6 - Value of c is 11
Line 7 - Value of c is 10
```

3.17.2 Relational Operators

- We often compare two quantities and depending on their relation, take certain decision. These comparisons can be done with the help of relational operator.
- An expression containing a relational operator is termed as a relational expression. The value of a relational expression is either one or zero. It is one if the specified relation is true and zero if the relation is false.
- Assume variable A holds 10 and variable B holds 20, then:

Operator	Description	Example
= =	Checks if the values of two operands are equal or not, if yes then condition becomes true.	(A == B) is not true.
!=	Checks if the values of two operands are 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.

3.17.3 Logical Operators

- C has the following three logical operators.

  && Meaning logical AND
  || Meaning logical OR
  ! Meaning logical NOT

- The logical operator && and || are used when we want to test more than one condition and make decision. For Example

  a>b && x = = 10

- An expression of this kind, which combines two or more relational expressions, is termed as a logical expressions or a compound relational expressions.
- Like the simple relational expressions, a logical expression also yields a value of one or zero.

Operands		Results			
x	y	!x	!y	x && y	x || y
0	0	1	1	0	0
0	Non-zero	1	0	0	1
Non-zero	0	0	1	0	1
Non-zero	Non-zero	0	0	1	1

3.17.4 Assignment Operators

- Assignment operator are used to assign the result of an expression to a variable.

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

3.17.5 Increment and decrement Operators

- The **increment operator (++)** causes its operand to be increased by 1 whereas the **decrement operator (--)** causes its operand to be decreased by 1. The operand used with each of these operators must be a single variable.
- For Example :-

```
x++;    /* equivalent to x = x + 1; */
```

- ++ and -- can be used in prefix or postfix notation. In prefix notation the value of the variable is either incremented or decremented and is then read while in postfix notation the value of the variable is read first and is then incremented or decremented.
- For Example :-

```
int i, j = 2 ;
i = ++ j  ;      /* prefix  :-  i has value 3, j has value 3  */
i = j++ ;        /* postfix  :-  i  has value 3, j has value 4   */
```

3.17.6 Bitwise Operators

- These are special operators that act on **char or int arguments only**. They allow the programmer to get closer to the machine level by operating at bit-level in their arguments.

&	Bitwise AND	|	Bitwise OR
^	Bitwise XOR	~	Ones Complement
>>	Shift Right	<<	Shift left

- Recall that type char is one byte in size. This means it is made up of 8 distinct bits or binary digits normally designated as illustrated below with Bit 0 being the Least Significant Bit (LSB) and Bit 7 being the Most Significant Bit (MSB). The value represented below is 13 in decimal.

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
0	0	0	0	1	1	0	1

An integer on a 16 bit OS is two bytes in size and so Bit 15 will be the MSB while on a 32 bit system the integer is four bytes in size with Bit 31 as the MSB.

- The bitwise operators are most commonly used in system level programming where individual bits of an integer will represent certain real life entities which are either on or off, one or zero. The programmer will need to be able to manipulate individual bits directly in these situations.
- A *mask* variable which allows us to ignore certain bit positions and concentrate the operation only on those of specific interest to us is almost always used in these situations. The value given to the mask variable depends on the operator being used and the result required.
- For Example: - To clear bit 7 of a char variable.

```
char ch = 89 ;          // any value
char mask = 127 ;       // 0111 1111
ch = ch & mask ;        // or  ch &= mask ;
```

- For Example: - To set bit 1 of an integer variable.

```
int i = 234 ;           // any value
int mask = 2 ;          // a 1 in bit position 2
i|= mask ;
```

Bitwise AND, &

RULE: If any two bits in the same bit position are set then the resultant bit in that position is set otherwise it is zero.

For Example:-

```
        1011 0010      (178)
&       0011 1111      (63)
=       0011 0010      (50)
```

Bitwise OR, |

RULE: If either bit in corresponding positions are set the resultant bit in that position is set.

For Example:-

```
        1011 0010          (178)
|       0000 1000          (63)
=       1011 1010          (186)
```

Bitwise XOR, ^

RULE: If the bits in corresponding positions are different then the resultant bit is set.

For Example:-

```
        1011 0010        (178)
^       0011 1100        (63)
=       1000 1110        (142)
```

Shift Operators, << and >>

RULE: These move all bits in the operand left or right by a specified number of places.

Syntax:
```
variable << number of places
variable >> number of places
```

For Example:-

```
2 << 2 = 8
```

i.e.

0000 0010 becomes 0000 1000

Note: shift left by one place multiplies by 2
shift right by one place divides by 2

Ones Complement

RULE: Reverses the state of each bit.

For Example:-

1101 0011 becomes 0010 1100

Note : With all of the above bitwise operators we must work with decimal, octal, or hexadecimal values as binary is not supported directly in C.

3.17.7 Conditional operator (?:)

- This operator is used for moving two way decision
- It takes three operands
- **Syntax:** conditional expression?expression-1:expression-2
- When evaluating a conditional expression is evaluated first. If conditional expression is true, the value of expression1 is the value of conditional expression. If conditional expression is false, the value of expression2 is the value of conditional expression.

Example:

```
a = 10;
b = 15;
x = (a>b)?a:b;
```

In this example, x will be assigned the value of b.

3.17.8 Special Operators

- C supports some special operators such as comma operator, size of operator, pointer operator (* and &) and member selection operators. (. and→).

❖ Comma Operator:

- The comma operator can be used to link the related expression together. A comma linked list of expression are evaluated left to right and the value of right-most expression is the value of combined expression. For example,

  value = (x = 10, y = 5, x + y),

  Here, 10 is assigned to x and 5 is assigned to y and so expression x+y is evaluated as (10+5) i.e. 15.

❖ sizeof Operator:

- The **sizeof** operator is used with an operand to return the number of bytes it occupies. It is a compile time operand. The operand may be a variable, a constant or a data type qualifier. The associativity of size of right to left.
- For example :

  Suppose that i is an integer variable, x is a floating-point variable, d is double-precision variable and c is character type variable. The statements:

```
printf ("integer : %d \n", sizeof(i));
printf ("float : %d \n", sizeof (x));
printf ("double : %d \n", sizeof (d));
printf ("character : %d \n", sizeof (c));
```

- The **sizeof** operator is normally used to determine the length of arrays and structures when their sizes are not known to the programmer. It is also used to allocate memory space dynamically to variables during execution of program.

3.18 Operator Precedence and Associativity

- If more than one operators are involved in an expression then, C language has predefined rule of priority of operators. This rule of priority of operators is called operator precedence.
- There are distinct levels of precedence and an operator may belong to one of these levels.
- The operators of higher precedence are evaluated first.
- Associativity indicates in which order two operators of same precedence (priority) executes.
- The operators of same precedence are evaluated from right to left or from left to right depending on the level.
- An **arithmetic expression** without parenthesis will be evaluated from left to right using the rules of precedence of operators.
- There are two distinct priority levels of arithmetic operators in C.
  - High priority * / %
  - Low priority + -
- Parentheses allow us to change the order of priority because of their highest precedence. When in doubt, we can add an extra pair just to make that the priority assumed is the one we required

Example: Find the value of result Assume x = 1, y = 2, z = 3 and p = 6

result= x + y * z / 2 + p;

```
#include<stdio.h>
#include<conio.h>
int main()
{
    int x=1,y=2,z=3,p=6,result;
    result= x + y * z / 2 + p;
    printf("Result = %d",result);
    getch();
    return 0;
}
```

x + y * z / 2 + p

x + (y * z) / 2 + p

x + ((y * z) / 2) + p

(x + ((y * z) / 2)) + p

((x + ((y * z) / 2)) + p)

Output:

Result = 10

- C operators are listed in order of precedence (highest to lowest). Their associativity indicates in what order operators of equal precedence in an expression are applied.

Operator	Description	Associativity
() [ ] . -> ++ --	Parentheses: grouping or function call Brackets (array subscript) Member selection via object name Member selection via pointer Postfix increment/decrement	left-to-right
++ -- + - ! ~ (*type*) * & sizeof	Prefix increment/decrement Unary plus/minus Logical negation/bitwise complement Cast (convert value to temporary value of *type*) Dereference Address (of operand) Determine size in bytes on this implementation	right-to-left
* / %	Multiplication/division/modulus	left-to-right
+ -	Addition/subtraction	left-to-right
<< >>	Bitwise shift left, Bitwise shift right	left-to-right
< <= > >=	Relational less than/less than or equal to Relational greater than/greater than or equal to	left-to-right
== !=	Relational is equal to/is not equal to	left-to-right
&	Bitwise AND	left-to-right
^	Bitwise exclusive OR	left-to-right
|	Bitwise inclusive OR	left-to-right
&&	Logical AND	left-to-right
||	Logical OR	left-to-right
? :	Ternary conditional	right-to-left
= += -= *= /= %= &= ^= |= <<= >>=	Assignment Addition/subtraction assignment Multiplication/division assignment Modulus/bitwise AND assignment Bitwise exclusive/inclusive OR assignment Bitwise shift left/right assignment	right-to-left
,	Comma (separate expressions)	left-to-right

4 Control Structures

4.1 Control Statements

- In C program, large number of functions are used that take data, process the data and return the result to the calling functions.
- A function is setup to perform a task. When the task is complex, many algorithms can be designed to achieve the same goal.
- Some algorithms may be simple but some may be complex. Experience has also shown that the number of bugs that occur is related to the format of the program.
- The format should be such that it is easy to trace the flow of execution of statements. This would help not only in debugging but also in the review and maintenance of the program later.
- One method of achieving the objective of an accurate, error-resistant and maintable code is to use one or any combination of control statements.

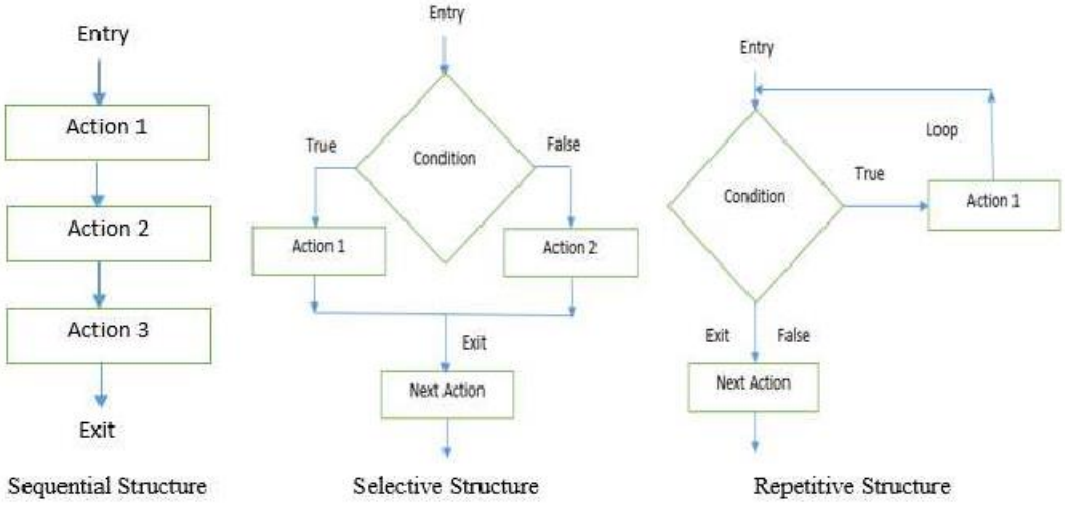


Figure 4-1: Basic Control Structure

- Figure 4.1 shows the execution of control structures using single-entry and single-exit concepts, which is a popular concept in modular programming.
- It is important to understand that entire program can be coded by using only these three logic structures.
- The approach of using one or more of these basic control structures in programming is known as structured programming.
- C supports all the three basic control structures and implements them using various control statements.

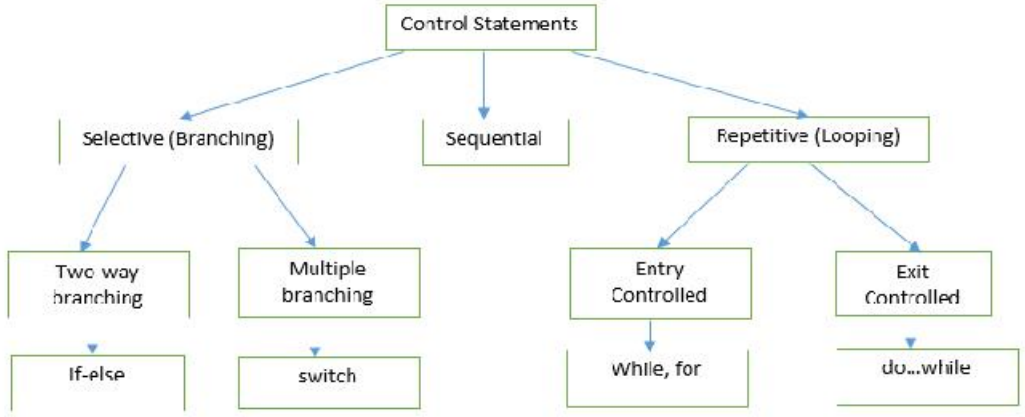


Figure 4-2: Classification of C Control Statements

4.2 Sequential Structure (Straight line flow)

- Sequential structure are the simplest structures in C.
- Sequence simple means executing one instructions after another, in the order in which they occur in the source file.
- This structured is used when no options and no repetition of certain calculations are necessary.
- There is no special statement for sequential structure.

Example 4-1: Write a program to calculate area and circumference of a circle. The program commands the user to enter the radius of circle.

```
#include<stdio.h>
#include<conio.h>
#define PI 3.1415
int main()
{
    float area,radious, circumf;                    /* Variable declaration*/
    printf("Enter radious of circle:");
    scanf("%f",&radious);
    area=PI*radious*radious;                        /*calculation of area*/
    circumf=2*PI*radious;                           /*calculation of circumference*/
    printf("\nArea of the circle= %f",area);
    printf("\nCircumference of circle= %f",circumf);
    getch();
    return 0;
}
```

4.3 Selective Structures (Branching)

- Selections means executing different sections of code depending on a condition or the value of a variable.
- This is what allows a program to take different courses of action depending on different circumstances.
- Selective structure are used when we have a number of situations where we may need to change the order of execution of statements based on certain conditions.
- The selective statements must make a decision to take the right path before changing the order of execution.
- C provides the following statements for selective structures
    - if statement
    - switch statement

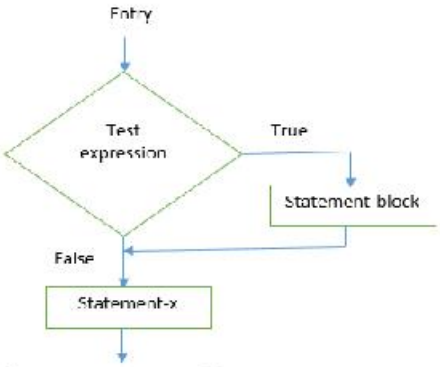
4.3.1 Simple if Statement

- The simple if statement is used to conditionally execute a block of code based on whether a test condition is true or false.
- If the condition is true the block of code is executed, otherwise it is skipped

Syntax:

```
if(test condition)
{
        Statement-block;
}
Statement-x;
```

Flowchart:



- Statement-block may be a single statement or a group of statements or nothing.
- If the test expression is true the statement-block will be executed otherwise the statement-block will be skipped and the execution will jump to statement-x. But when the condition is true both the statement-block and statement-x are executed in sequence.

Example 4-2: Write a program that prompts a user to input average marks of a student and adds 10% bonus marks if his/her average marks is greater than or equal to 65%.

```
#include<stdio.h>
#include<conio.h>
int main()
{
        float marks;
        printf("Enter marks:");
        scanf("%f",&marks);
        if(marks>=65)
        {
                marks=marks+marks*0.1;
        }
        printf("Final marks=%f",marks);
        getch();
        return 0;
}
```

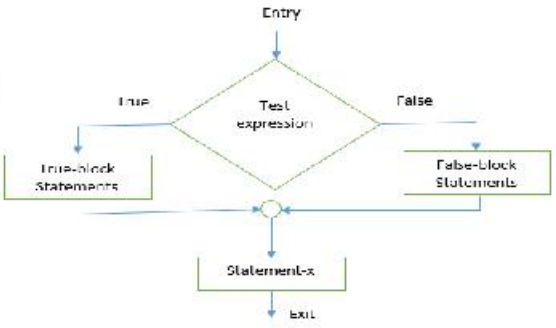
4.3.2 if-else Statement

- The if…else statements is an extension of simple if statement.
- If the test expression is found to be true it will execute the if block statements. If the test expression is found to be false, it will skip true block statement and execute the false block statement i.e. else block statement are executed.

Syntax:

```
if(test expression)
{
        True-block statements;
}
else
{
        False-block Statements;
}
```

Flowchart:



Example 4-3: write a program to compare two numbers

```
#include<stdio.h>
#include<conio.h>
int main()
{
        int num1, num2;
        printf("Enter first number");
        scanf("%d",&num1);
        printf("Enter second number");
        scanf("%d",&num2);
        if(num1>num2)
        {
                printf("%d is greater than %d",num1,num2);
        }
        else
        {
                printf("%d is not greater than %d",num1,num2);
        }
        getch();
        return 0;
}
```

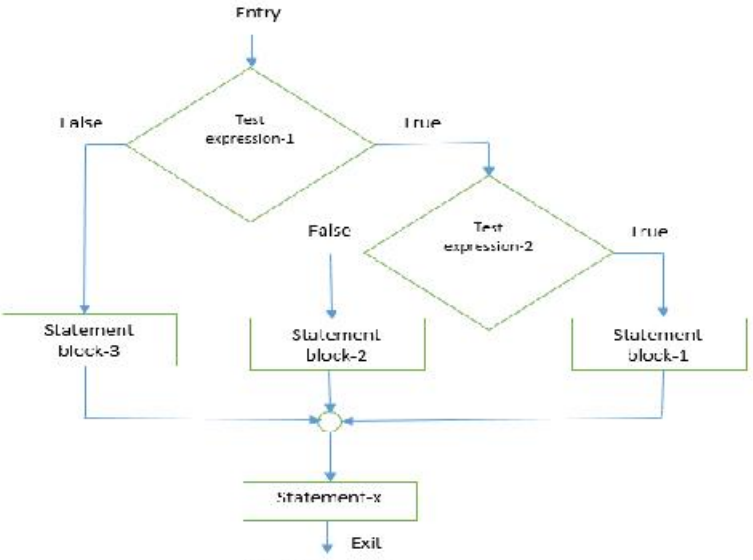
4.3.3 Nested if…else Statement

- More than one if…else statements is used in nested form to solve a series of decision.

Syntax:

```
if(test expression-1)
{
        if(test-expression-2)
        {
                Statement block-1;
        }
        else
        {
                Statement block-2;
        }
}
else
{
        Statement block-3;
 }
```

Flowchart:



Example 4-4: Write a program to find largest of three numbers

```
#include<stdio.h>
#include<conio.h>
int main()
{
        int a,b,c;
        printf("Enter three numbers:");
        scanf("%d%d%d",&a,&b,&c);
        if(a>b)
        {
                if(a>c)
                printf("The largest number is %d",a);
                else
                printf("The largest number is %d",c);
        }
        else
        {
                if(b>c)
                printf("The largest number is %d",b);
                else
                printf("The largest number is %d",c);
        }
        getch();
        return 0;
}
```

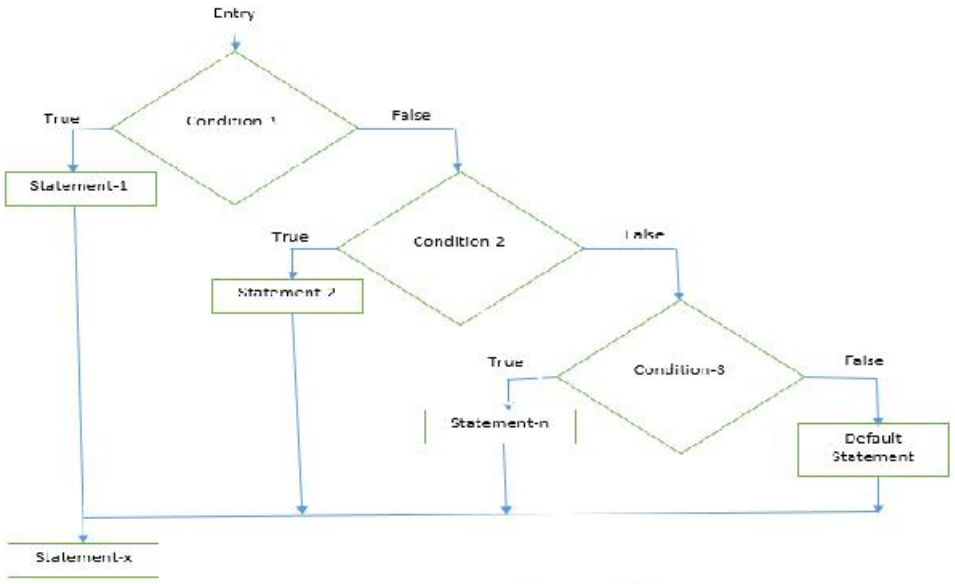
4.3.4 The else if Ladder

- It is another way of putting ifs together when multiple decision are involved
- A multipath decision is a chain of ifs in which the statement associated with each else is if.

Syntax:

```
if(condition-1)
        statement-1;
else if(condition-2)
        statement-2;
……….
else if(condition-n)
        Statement-n;
else
        Default statement;
Statement-x;
```

Flowchart:



- The test expression are evaluated from the top to downward. As soon as a true condition is found the statement associated with it is executed and the rest of the ladder is bypassed and control is transferred to the statement-x.
- If non of the condition is true then the final else statement will be executed. The final else often act as a default condition i.e. if all other condition's test fail then the last else statement is executed.
- If there is no final else and all other conditions are false then no action will take place.
-

Example 4-5: This example prompts the user to enter any integer from 1 to 7 and display the corresponding day of the week

```
#include<stdio.h>
#include<conio.h>
int main()
{
        int d;
        printf("Enter an integer");
        scanf("%d",&d);
        if(d==1)
                printf("Sunday");
        else if(d==2)
                printf("Monday");
        else if(d==3)
                printf("Tuesday");
        else if(d==4)
                printf("Wednesday");
        else if(d==5)
                printf("Thursday");
        else if(d==6)
                printf("Friday");
        else if(d==7)
                printf("Saturday");
        else
                printf("Enter only 1 to 7");
        getch();
        return 0;
}
```

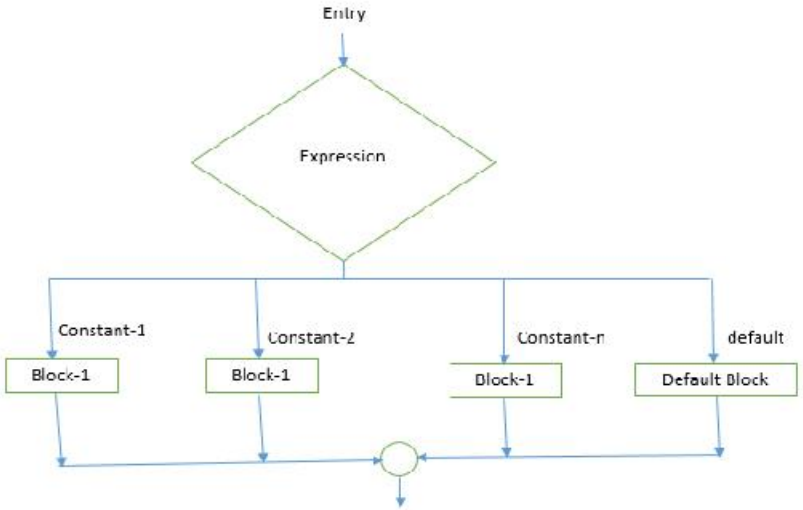
4.3.5 The switch case statement

- The switch case statement successively tests the value of a given variables (an expression) with a list of case values (integer or character constants). When a match is found, the statement associated with that case is executed.

Syntax:

```
switch(expression)
{
        case constant-1:
                block-1;
                break;
        case constant-2:
                block-2;
                break;
        ……..
        default:
                default-block;
}
Statement-x;
```

Flowchart:



- **Switch statement behavior**

Condition	Action
• The value of switch expression is matched to a case label. • The value of switch expression is not matched to any case label and the default label is present • The value of switch expression is not matched to any case label and the default label is not present	• Control is transferred to the statement following that label. • Control is transferred to the statement following default label. • Control is transferred to the statement after the switch statement.

Note:

- Expression can be integer or character.
- Constant-1, constant-2… are integral constant or character.
- When the switch is executed, the value of expression successfully compared against the values constant-1, constant-2…the matching case is then activated and its block is executed.
- break passes control out of switch
- default optional case

Example 4-6: Write a program that asks an arithmetic operator and two operands and performs the corresponding operations.

```
#include<stdio.h>
#include<conio.h>
int main()
{
        char o;
        int a,b;
        float  result;
        printf("Enter value of a:");
        scanf("%d",&a);
        printf("Enter value of b:");
        scanf("%d",&b);
        printf("Enter a arithmetic operator + - * or /");
        fflush(stdin);
        scanf("%c",&o);
        switch(o)
        {
                case '+':
```

```
                result= a+b;
                printf("\n a+b=%f",result);
                break;
        case '-':
                result= a-b;
                printf("\na-b=%f",result);
                break;
        case '*':
                result= a*b;
                printf("\na*b=%f",result);
                break;
        case '/':
                result= a/b;
                printf("\na/b=%f",result);
                break;
        default:
                printf("Enter only + - * or /");
    }
    getch();
    return 0;
}
```

❖ Difference between else..if and switch statement

else..if	switch
• An expression is evaluated and the code is selected based on the truth-value of the expression	• An expression is evaluated and the code is selected based on the value of expression
• Each if has its own logical expression to be evaluated as true or false	• Each case is referring back to the original value of the expression in the switch statement
• The variable in the expression may evaluate to a value of any type, either an int or a char	• The expression must evaluated to an int
• It does not require break statement because only one of the blocks of code is executed	• It needs involvement of the break statement to avoid execution of the block just below the current executing block
• It takes decision on the basis of nonzero (true) or zero (false) basis.	• It takes decision on the basis of equality

❖ The Conditional operator (?:)

- This operator is used for moving two way decision
- It takes three operands
- **Syntax:**
  conditional expression?expression-1:expression-2

Example:

```
if(a>b)
        value=2*a-b;
else
        value=a+5*b;
```

can be written as

```
value=(a>b)?(2*a-b):(a+5*b);
```

- The conditional operator may be nested for evaluating more complex assignment decision

Example 4-7: Write a program that finds the value of y based on the value of x in the following function

$$y = \begin{cases} 2x + 300 & for\ x < 50 \\ 200 & for\ x = 50 \\ 50x - 100 & for\ x > 50 \end{cases}$$

```
#include<stdio.h>
#include<conio.h>
int main()
{
        float x,y;
        printf("Enter value of x:");
        scanf("%f",&x);
        y=(x!=50)?((x<50)?(2*x+30):(50*x-100)):200;
        printf("Value of y is %.2f",y);
        getch();
        return 0;
}
```

4.4 Repetitive Structure (iteration or loop structure)

- Looping is a process in which the sequence of statements may be executed for a specified number of times or until some condition is met.
- The structures which executes the statements for a specified number of times are called count controlled loops. The structures which executes the statements until some condition is met are called condition (sentinel) controlled loops.
- A looping process, in general would include the following four steps
  - Setting and initialization of a counter
  - Execution of statements in the loop
  - Test for a specified condition for execution of the loop
  - Updating the counter
- A program loop consist of two segments
  - **Body of the loop:** are the statements that have to be executed repeatedly.
  - **Control Statement:** test for a given condition.
- Depending upon the position of the control structure, it may be classified as either an entry control loop or an exit control loop.
  - In an entry controlled loop the control conditions are tested before the start of the loop execution. If the condition are not satisfied the body of the loop will not be executed. This loop is also known as pretest loop.
  - In case of exit controlled loop the test is performed at the end of the body of the loop and therefore the body is executed unconditionally for the first time.

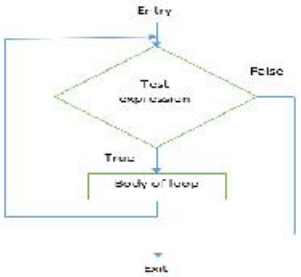
4.4.1 The while statement

- The while statement specifies that a section of code should be executed while a certain condition holds true.

Syntax:

```
while(test expression)
{
    body of loop
}
```

Flowchart:



Example 4-8: WAP to read numbers continuously and sum them till sum is less than 100.

```
#include<stdio.h>
#include<conio.h>
int main()
{
```

```
    int n,sum=0;
    while(sum<100)
    {
        printf("Enter a number:");
        scanf("%d",&n);
        sum=sum+n;
    }
    printf("Sum=%d",sum);
    getch();
    return 0;
}
```

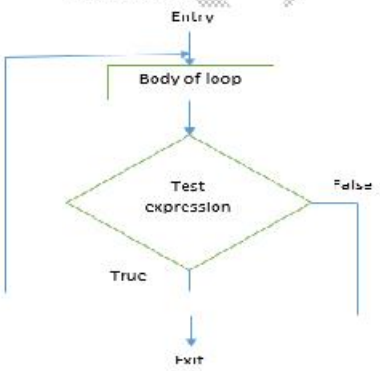
4.4.2 do…while statement

- The do…while statement is very similar to the while statement.
- On reaching the do…while loop, the program proceeds to evaluate the body of loop first, at the end of loop the test expression in the while statement is evaluated. If the expression evaluates to true, the program continue to evaluate the body of the loop once again. This process continues as long as the condition is true. When the condition become false, the loop will be terminated and the control goes to the statement that appears immediately after the while statement.
- The difference between a do…while loop and a while loop is that the while loop tests its condition at the top but the do…while loop tests its condition at the bottom of its loop.
- The do…while loop executes the program at least one time either the condition is true or false.

Syntax:

```
do
{
    Body of loop;
}while(test expression);
```

Flowchart:



Example 4-9: Calculate the sum and average of n numbers

```
#include<stdio.h>
#include<conio.h>
int main()
{
    int n,count=1;
    float x,average,sum=0;
    printf("How many numbers?");
    scanf("%d",&n);
    do{
        printf("Enter %d number",count);
        scanf("%f",&x);
        sum+=x;
        ++count;
    }while(count<=n);
    average=sum/n;
    printf("\nThe Sum is %f",sum);
    printf("\nThe average is %f",average);
    getch();
    return 0;
}
```

4.4.3 The for Statement

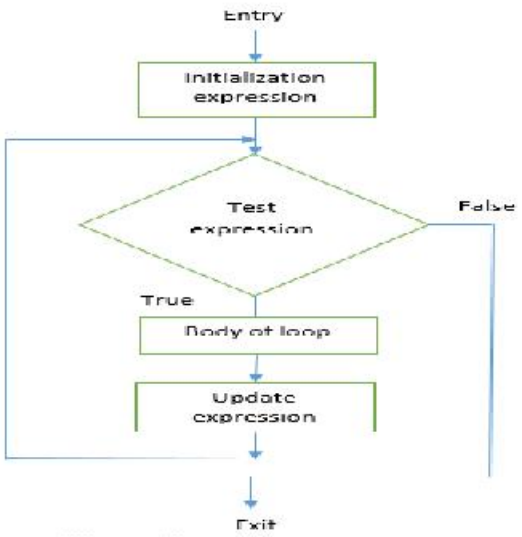
- Both the while and do…while statements provides repetition until some condition become false.
- The for statement uses a loop variable to count the number of times that the loop has been executed.

Syntax:

```
for( initialization expression; test expression; update expression)
{
        body of loop
}
```

- Initialization is generally an assignment statement used to set the loop control variable.
- Test expression is a relational expression that determines when the loop exits.
- Update expression defines how the loop variable changes each time the loop is repeated.

Flowchart:



Example 4-10: WAP to print numbers from 0 to 10.

```
#include<stdio.h>
#include<conio.h>
int main()
{
        int i;
        for(i=0;i<=10;i++)
        {
            printf("\n%d",i);
        }
        getch();
        return 0;
}
```

❖ Variations of for loop

- **More than one variable can be initialized at a time using comma operator**

```
i=1;
for(j=0;j<10;j++)
```

can be written as

```
for(i=1,j=0;j<10;j++)
```

- **There may be more than one part in update section as in the initialization section separated by comma operator**

```
for(i=0,j=100;i<=j;i++,j--)
{
        sum= i+j;
        printf("sum=%d",sum);
}
```

- **Test condition may have any compound relational and the testing need not be limited only to the loop control variable**

```
 sum=0;
for(i=0;i<50&&sum<500;i++)
{
        sum=sum+i;
        printf("%d\n",sum);
}
```

- **Missing pieces of the loop definition**
  - **Omission of initialization expression**

```
i=0;
for(;i<30;i++)
```

  - **Omission of initialization and test condition**

```
i=0;
for(;;i++)
{
        if(i>20)
        break;
        printf("nec");
}
```

  - **Omission of initialization, test and update expression**

```
for(;;;)
{
        ch=getchar();
        if(ch=='A');
        break;
}
```

    - An infinite loop is created
  - **Omission of initialization and update expression**

```
i=0;
for(;i<30;)
{
        prinf("%d\n",i);
        i++;
}
```

- **for loops with no body (null statement)**

```
for(t=100;t>=0;t--);
```

4.4.4 Nesting of loops

- Putting one loop statement within another loop statement is called nesting of loops

❖ **Nesting of for loops**

```
for(i=0;i<10;i++)
{
        ……..
        for(j=0;j<10;j++)
        {
                ……..
        }
        ……..
}
```

❖ **Nesting of while loops**

```
i=0;
while(i<=rows)
{
        j=0;
        while(j<=columns)
        {
                product=i*j;
                printf("%d",product);
                j++;

        }
        i++;
}
```

❖ **Nesting of do…while loops**

```
i=0;
do
{
        j=0;
        do
        {
                product=i*j;
                printf("%d",product);
                j++;

        }while(j<=columns);

        i++;

}while(i<=rows);
```

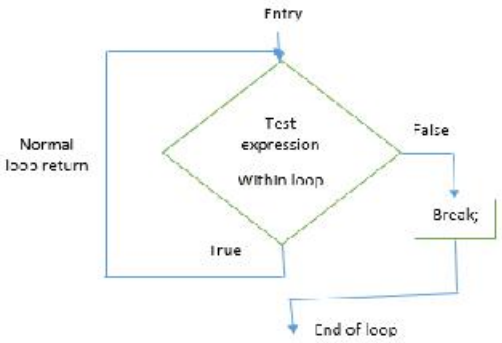
4.5 Jumps in loops

- Loops perform a set of operations repeatedly until the control variable fails to satisfy the test condition. The number of times a loop is repeated is decided in advance and the test condition is written to achieve this. Sometimes, when executing a loop it becomes desirable to skip a part of the loop or to leave the loop as soon as a certain condition occurs.
- C permits a jump from one statement to another within loop as well as a jump out of loop.
- break, continue, goto and return statements are used for jumping purpose.

4.5.1 The break statement

- break statement is used to jump out of a loop
- it terminates the execution of the nearest enclosing loop.

Syntax:
break;



Example 4-11: WAP to read only positive integer numbers and sum them until sum is less than equal to 100.

```
#include<stdio.h>
#include<conio.h>
int main()
{
      int n,sum=0;
      while(sum<=100)
      {
            printf("Enter a number(Value of n):");
            scanf("%d",&n);
            if(n>=0)
            sum=sum+n;
            else
            break;
      }
      printf("Sum=%d",sum);
      getch();
      return 0;
}
```

```
while(...)
{
      .....
      if(condition)
            break;
      .....
}
.....
```

Exit from loop

```
do
{
      .....
      if(condition)
            break;
      .....
}while(...)
.....
```

Exit from loop

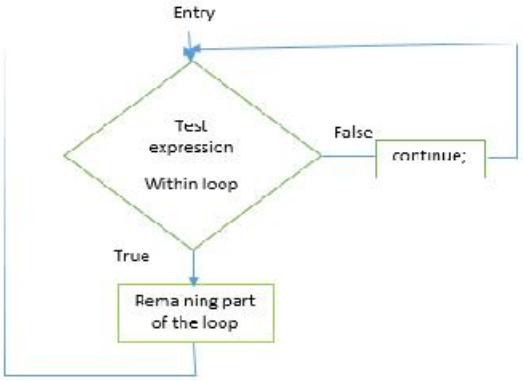
```
for(...)
{
      for( )
      {
            .....
            if(condition)
            break;
            .....
      }
}
```

Exit from loop

4.5.2 The continue statement

- continue statement is used to bypass the remainder of the current pass through a loop
- The loop does not terminate when a continue statement is encountered. The remaining loop statement are skipped and the computation proceeds directly to the next pass through the loop.

Syntax:
continue;



Example 4-12: WAP to read integer numbers and sum only positive numbers until sum is less than equal to 100.

```
#include<stdio.h>
#include<conio.h>
int main()
{
        int n,sum=0;
        while(sum<=100)
        {
                printf("Enter a number(Value of n):");
                scanf("%d",&n);
                if(n>=0)
                sum=sum+n;
                else
                continue;
        }
        printf("Sum=%d",sum);
        getch();
        return 0;
}
```

Example 4-13: WAP to display ASCII value of all alphabets

```
#include<stdio.h>
#include<conio.h>
int main()
{
        int i;
        for(i=65;i<=122;i++)
        {
                if(i>90&&i<97)
                continue;
                printf("%c=%d\t",i,i);
        }
        getch();
        return 0;
}
```

4.5.3 The goto statement

- The goto statement is used to alter the normal sequence of program execution by transferring control to some other parts of the program.

Syntax:

goto label;	label:
.....	.....
.....	.....
label:	goto label;
Statement;	statement;
a) Forward jump	b) backward jump

- Where label is an identifier that is used to label the target statement to which the control will be transferred.
- The target statement must be labeled and the label must be followed by a colon (:) whereas the label followed by goto statement must be terminated by semicolon (;).
- Each label statement within program (more precisely within the current function) must have unique label.
- If the label: is placed after goto label; statements then the some statements will be skipped and the jump is known as a forward jump.
- On the other hand if the label: is before the statement goto label; a loop will be formed and some statement will be executed repeatedly such a jump is known as backward jump.

Example 4-14: WAP to sum all numbers from 0 to 100.

```
#include<stdio.h>
#include<conio.h>
int main()
{
        int i=0,sum=0;
        label:
                sum=sum+i;
                i++;
                if(i<=100)
                goto label;
                printf("Sum=%d",sum);
        getch();
        return 0;
}
```

Some Solved Example

1. **Write a C Program to find the roots of a quadratic equation when coefficients are entered by user.**

```
#include <stdio.h>
#include<conio.h>
#include <math.h>
int main()
{
        float a, b, c, determinant, r1,r2, real, imag;
        printf("Enter coefficients a, b and c: ");
        scanf("%f%f%f",&a,&b,&c);
        determinant=b*b-4*a*c;
        printf("determinant=%f\n",determinant);
       if (determinant>0)
       {
              r1= (-b+sqrt(determinant))/(2*a);
              r2= (-b-sqrt(determinant))/(2*a);
              printf("Roots are: %.2f and %.2f",r1 , r2);
       }
       else if (determinant==0)
       {
              r1 = r2 = -b/(2*a);
              printf("Roots are: %.2f and %.2f", r1, r2);
       }
       else
       {
              real= -b/(2*a);
              imag = sqrt(-determinant)/(2*a);
              printf("Roots are: %.2f+%.2fi and %.2f-%.2fi", real, imag, real, imag);
       }
        getch();
        return 0;
}
```

2. **Write a program to read three sides of triangle and print area for valid data and to print "Invalid data" if either one side of the triangle is greater or equals to the sum of other two sides.**

```
#include <stdio.h>
#include <conio.h>
#include <math.h>
int main()
{
       int a,b,c;
       float area,s;
       printf("Enter three side of triangle:");
       scanf("%d%d%d",&a,&b,&c);
       if(a>=(b+c) || b>=(a+c) || c>=(a+b))
       {
              printf("Invalid data");
       }
       else
       {
              s=(a+b+c)/2.0;
              area= sqrt(s*(s-a)*(s-b)*(s-c));
              printf("Area=%f",area);
       }
```

```
        getch();
        return 0;
    }
```

3. **Write a program to read three numbers and display the following menu.**

Menu:	**Press**
Summation	**1**
Sum of squares	**2**
Sum of cubes	**3**
Product	**4**

and perform tasks as per user's choice. (use switch statements)

```
#include<stdio.h>
#include<conio.h>
int main()
{
    int sw,a,b,c,result;
    printf("Enter three number:");
    scanf("%d%d%d",&a,&b,&c);
    printf("Enter choice\n 1.summation\n 2.sum of squares\n 3.sum of cubes\n 4.product\n");
    scanf("%d",&sw);
    switch(sw)
    {
        case 1:
            result=a+b+c;
            printf("Summation=%d",result);
            break;
        case 2:
            result=(a*a)+(b*b)+(c*c);
            printf("Sum fo squares=%d",result);
            break;
        case 3:
            result=(a*a*a)+(b*b*b)+(c*c*c);
            printf("Sum of cubes=%d",result);
            break;
        case 4:
            result=a*b*c;
            printf("Product=%d",result);
            break;
        default:
            printf("Invalid Choice");
    }
    getch();
    return 0;
}
```

4. **Write a program to read a character and to test whether it is an alphabet or a number or a special character.**

```
#include<stdio.h>
#include<conio.h>
int main()
{
    char ch;
    printf("Please enter a character : ");
    scanf("%c", &ch);
    if(('a' <= ch && ch <= 'z') || ('A' <= ch && ch <= 'Z'))
    {
        printf("Entered character %c is a alphabet", ch);
    }
```

```
        else if('0' <= ch && ch <= '9')
        {
                printf("Entered character %c is a number", ch);
        }
        else
        {
                printf("Entered character %c is a special character", ch);
        }
        return 0;
        getch();
}
```

5. Write a C program to find and print Armstrong numbers in range between two numbers given by a user.
[Hint: 153 is Armstrong number because $1^3+5^3+3^3=153$]

```
#include<stdio.h>
#include<conio.h>
int main()
{
        int m,n,r,i,k,sum;
        printf("Enter starting number:");
        scanf("%d", &m);
        printf("Enter ending number:");
        scanf("%d", &n);
        for(i=m;i<=n;i++)
        {
                k=i;
                sum=0;
                do{
                        r=k%10;
                        sum=sum+r*r*r;
                        k=k/10;
                }while(k!=0);
                if(sum==i)
                {
                        printf("\t%d",i);
                }
        }
        getch();
        return 0;
}
```

6. Write a C program to find out octal equivalent when a decimal integer number is given.

```
#include<stdio.h>
#include<conio.h>
#include<math.h>
int main()
{
    int x,i=0,sum=0,R;
    printf("Enter any integer number:");
    scanf("%d",&x);
    while(x!=0)
    {
        R=x%8;
        sum=sum+R*pow(10,i);
        x=x/8;
```

```
        i++;
      }
    printf("The octal equivalent to given desimal integer = %d",sum);
    getch();
    return 0;
  }
```

7. Write a C program to generate the following series and print sum.
1 x 4, 2 x 7, 3 x 10, …… n terms

```
#include<stdio.h>
#include<conio.h>
int main()
{
        int i,j,n,sum=0;
        printf("Enter number of term:");
        scanf("%d",&n);
        for(i=1,j=4;i<=n;i++,j+=3)
        {
                printf("%dx%d,\t",i,j);
                sum=sum+(i*j);
        }
        printf("\nSum of series=%d",sum);
        getch();
        return 0;
}
```

8. Write a program to read x and n, and generate the following series & print the sum.
$x + 2/x^2 - 3/x^3 + 4/x^4 - 5/x^5$……n terms

```
#include<stdio.h>
#include<conio.h>
#include<math.h>
int main()
{
    int i,x,n;
    float a,sum=0;
    printf("Enter the value and number of term\n");
    scanf("%d%d",&x,&n);
    if(n==0)
            printf("There is no term");
    if(n==1)
            printf("The term is:\n %d",x);
    if(n>=2)
    {
            printf("The term is:\n%d",x);
            for(i=2;i<=n;i++)
            {
                    a=(pow(-1,i)*i)/(pow(x,i));
                    printf("\t%f",a);
                    sum=sum+a;
            }
        printf("\nThe sum of the terms = %f",sum+x);
    }
   getch();
   return 0;
}
```

9. Write a C program to print first 50 prime numbers.

```
#include<stdio.h>
#include<conio.h>
int main()
{
        int i,j,count=0,n;
        printf("Enter the first value:");
        scanf("%d",&n);
        printf("The First 50 Prime Numbers from given value are:\n");
        for(i=n;count<50;i++)
        {
            for(j=2;j<=i;j++)
            {
                if(i%j==0)
                    break;
              }
            if(j==i)
            {
                count++;
                printf("%d\t",i);
                }
            }
        getch();
        return 0;
    }
```

10. An electricity board charges according to the following rates:

For the first 20 units----------Rs 80
For the next 80 units----------Rs 7.8 per unit
For the next 100 units----------Rs 8.5 per unit
For the Beyond 200 units----------Rs 9.5 per unit

And Tax 15% in total amount is charged to all users. Write a program to read number of units consumed and print out the total charges.

```
#include<stdio.h>
#include<conio.h>
int main()
{
    int units;
    float amount, total;
    printf("Enter number of units consumed:");
    scanf("%d",&units);
    if(units<=20)
    {
        amount=80;
    }
    else if(units<=100)
    {
        amount=80+(units-20)*7.5;
    }
    else if(units<=200)
    {
        amount=80+80*7.5+(units-100)*8.5;
    }
    else
    {
        amount=80+80*7.5+100*8.5+(units-200)*9.5;
```

```
        }
        total=amount+amount*0.15;
        printf("Total charges = Rs %f",total);
        getch();
        return 0;

    }
```

11. Write a program to find the HCF and LCM of two integer numbers just entered by user.

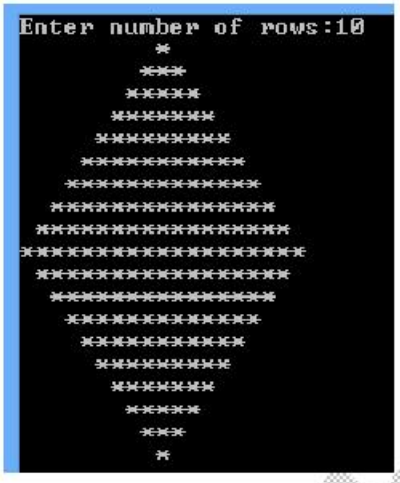
```
#include<stdio.h>
#include<conio.h>
int main()
{
    long firstnum,secondnum,remainder,temp;
    printf("Enter two integer numbers:");
    scanf("%ld%ld",&firstnum,&secondnum);
    temp=firstnum*secondnum;
    do
    {
        remainder=firstnum%secondnum;
        if(remainder==0)
        {
            printf("HCF=%ld",secondnum);
        }
        else
        {
            firstnum=secondnum;
            secondnum=remainder;
        }
    }while(remainder!=0);
    printf("\nLCM=%ld",temp/secondnum);
    getch();
    return 0;
}
```

12. Write a C program to print diamond pattern of star(*) using nested for loop.

```
#include<stdio.h>
#include<conio.h>
int main()
{
    int n,i,j, space;

    printf("Enter number of rows:");
    scanf("%d", &n);
    space=n-1;
    for (i=1;i<=n;i++)
    {
        for (j=1;j<=space;j++)
            printf(" ");
        for (j=1;j<=2*i-1;j++)
            printf("*");
```

```
                printf("\n");
                space--;
        }
        space=1;
        for (i=1;i<=n;i++)
        {
                for (j=1;j<=space;j++)
                        printf(" ");
                for (j=1;j<=2*(n-i)-1;j++)
                        printf("*");
                printf("\n");
                space++;
        }
        getch();
        return 0;
}
```

Output:



5 Arrays and Strings

5.1 Introduction to Array

- An array is collection of similar data items that share a common name.
- The items are called the elements of an array. The elements share the same variable name but each element has an index number also known as subscript.
- The first element of the array is referred with subscript 0(zero) & the subscript of last element is one less than size of an array i.e. (n-1).
- Array can be of any data type supported by C like char, int, float etc.
- An array can be single dimensional or multidimensional. The number of subscript give the dimension of an array.
- Consider the following example:

  int x,y,z;
  x=10;y=20;z=30;
  the above statements can be rewritten using array form
  int x[3];
  x[0]=10;x[1]=20;x[2]=30;

5.2 One-dimensional array

- Elements of the array can be represented either as a single column or a single row. While dealing with one dimensional arrays, we specify a single subscript to locate a specific elements.

5.2.1 Declaring one dimensional array

Syntax:

data-type array-name[size];

- The 'data-type' specifies type of the array. That is, int, char, float etc. The 'variable-name' is the name given to the array, by which we will refer it. Finally, size is the maximum number of elements that an array can hold.
- The size is specified by a positive integer expression, enclosed in square brackets. The size can be any +ve integer constant or any expression giving an integer.
- Example:

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

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

5.2.2 Initialization of array

- Array initialization can be done in following ways
  - int m[5]={1,2,3,8,9};
  - int m[]={1,2,3,4,5,6,7};
  - int m[10]={1,2,3,4}; in this first four elements are assigned 1,2,3,4 and remaining has zero value.

Syntax:

Data-type array-name [size] = {value1, value2, ……..valuen};

Note: int a[2]={1,2,3}; Invalid

5.2.3 Accessing array elements

- We can access the elements of an array by giving the name and proper subscript inside bracket[].
- E.g.

  float marks[5]={90,45,60.5,85,25};

address	1000	1004	1008	1012	1016
marks	90	45	60.5	85	25
element	marks[0]	marks[1]	marks[2]	marks[3]	marks[4]

- marks[0] is 90 which is stored at 1000
  i.e. marks[0]=90, marks[1]=45, marks[2]=60.5, marks[3]=85, marks[4]=25
- marks holds the address of marks[0].

Example 5-1: WAP that initialize five integer numbers into an array and displays that on the screen.

```
#include<stdio.h>
#include<conio.h>
int main()
{
        int a[5]={ 1,2,3,4,6},i;
        printf("The array elements are\n");
        for(i=0;i<5;i++)
        printf("%d\t",a[i]);
        getch();
        return 0;
}
```

Output:

The array elements are
1 2 3 4 6

Example 5-2: Find the maximum of elements present in the array

```
#include<stdio.h>
#include<conio.h>
int main()
{
        int a[30],n,i,max=0;
        printf("Enter number of element in the array:");
        scanf("%d",&n);
        for(i=0;i<n;i++)
        {
                printf("Enter %d element:",i+1);
                scanf("%d",&a[i]);
        }
        for(i=0;i<n;i++)
        {
                if(max<a[i])
                max=a[i];
        }
        printf("\nThe maximum element present in the array is %d",max);
        getch();
        return 0;
}
```

Example5-3: WAP to arrange an array of elements in Ascending order [BUBBLE SORT].

```
#include<stdio.h>
#include<conio.h>
int main()
{
        int a[30],i,j,t,n;
        printf("Enter size of array:");
        scanf("%d",&n);
```

```
        for(i=0;i<n;i++)
        {
                printf("Enter %d elements:",i+1);
                scanf("%d",&a[i]);
        }
        printf("Array before sorting:\n");
        for(i=0;i<n;i++)
        {
                printf("%d\t",a[i]);
        }
        for(i=0;i<n;i++)
        {
                for(j=0;j<n-i-1;j++)
                {
                        if(a[j]>a[j+1])
                        {
                                t=a[j];
                                a[j]=a[j+1];
                                a[j+1]=t;
                        }
                }
        }
        printf("\nArray After sorting:\n");
        for(i=0;i<n;i++)
        {
                printf("%d\t",a[i]);
        }
        getch();
        return 0;

}
```

Output:

```
Enter size of array:5
Enter 1 elements:4
Enter 2 elements:2
Enter 3 elements:6
Enter 4 elements:1
Enter 5 elements:3
Array before sorting:
4       2       6       1       3
Array After sorting:
1       2       3       4       6
```

5.3 Multidimensional arrays

- Multidimensional arrays are defined in much the same manner as one-dimensional arrays, except that a separate pair of square brackets is required for each subscript. Thus, a two-dimensional array will require two pairs of square brackets, a three-dimensional array will require three pairs of square brackets, and so on.
- The general form of a multidimensional array is

  data-type array-name[s1][s2][s3]……[sn];

  where si is the size of the ith dimension.

5.4 Two-dimensional array

- A two dimensional array is a collection of data items, all of the same type, structured in two dimensions (referred to as rows and columns).
- It is also called as array of arrays. Many times it is required to manipulate the data in table format or in matrix format which contains rows and columns. In these cases, we have to give two dimensions to the array. That is, a table of 5 rows and 4 columns can be referred as,

table[5][4]

The first dimension 5 is number of rows and second dimension 4 is number of columns.

- In order to create such two dimensional arrays in C, following syntax should be followed.

  data-type array-name[rows][columns];

For example:

int table[5][4];

This will create total 20 storage locations for two dimensional arrays as,

	Columns			
Rows	[0][0]	[0][1]	[0][2]	[0][3]
	[1][0]	[1][1]	[1][2]	[1][3]
	[2][0]	[2][1]	[2][2]	[2][3]
	[3][0]	[3][1]	[3][2]	[3][3]
	[4][0]	[4][1]	[4][2]	[4][3]

5.4.1 Initializing two-dimensional arrays

- int marks[3][4] = { {65,85,75,50},{67,65,45,75},{35,5,60,50} };
  or
  int marks[3][4] = {65,85,75,50,67,65,45,75,35,5,60,50};
- int values [][3] = {1,2,3,4,5,6};

5.4.2 Accessing two-dimensional array elements

- To access two-dimensional array we need to use the concept of rows and columns since array elements are arranged in a tabular form. Therefore, to access any element we need to use two loops, one for rows and one for columns.

Example 5-4: Suppose there are five batsman and they played five cricket matches and they scored runs as:

	Match1	Match2	Match3	Match4	Match5
Batsman1	125	35	65	25	55
Batsman2	10	65	45	35	25
Batsman3	20	35	47	42	85
Batsman4	75	35	25	35	30
Batsman5	50	25	35	105	5

Write a C program to find the average runs scored by each batsman.

```
#include<stdio.h>
#include<conio.h>
int main()
{
    int runs[5][5]={{125,35,65,25,55},{10,65,45,35,25},{20,35,47,42,85},{75,35,25,35,30},{50,25,35,105,5}};
    int i,j,sum;
    float avg;
    for(i=0;i<5;i++)
    {
        sum=0;
        for(j=0;j<5;j++)
        {
            sum+=runs[i][j];
        }
        avg=sum/5.0;
        printf("Batsman%d has scored %d runs with average %.2f\n",i+1,sum,avg);
    }
    getch();
    return 0;
}
```

Output:

```
Batsman1 has scored 305 runs with average 61.00
Batsman2 has scored 180 runs with average 36.00
Batsman3 has scored 229 runs with average 45.80
Batsman4 has scored 200 runs with average 40.00
Batsman5 has scored 220 runs with average 44.00
```

Example 5-5: Write a program to find the sum of diagonal elements (trace) of matrix.

```
#include<stdio.h>
#include<conio.h>
int main()
{
    int matrix[10][10],i,j,sum=0,r,c;
    printf("Enter the size of Square matrix\n");
    scanf("%d%d",&r,&c);
    if(r==c)
    {
        printf("\nEnter the elements of matrix\n");
        for(i=0;i<r;i++)
        {
            for(j=0;j<c;j++)
            {
                printf("Enter matrix[%d][%d] element:",i,j);
                scanf("%d",&matrix[i][j]);
                if(i==j)
                sum+=matrix[i][j];
            }
        }
        printf("\nSum of main diagnoal elements is %d",sum);
    }

    else
    printf("\nRows and Columns not matched");
    getch();
    return 0;
}
```

Output:

```
Enter the size of Square matrix
2
2

Enter the elements of matrix
Enter matrix[0][0] element:1
Enter matrix[0][1] element:2
Enter matrix[1][0] element:3
Enter matrix[1][1] element:4

Sum of main diagnoal elements is 5
```

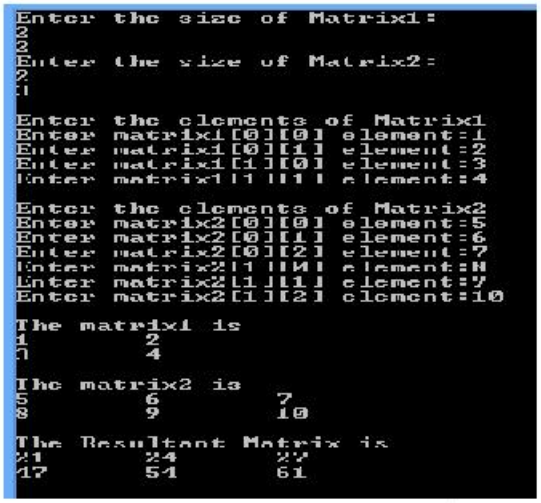
Example 5-6: Write a program to multiply two given matrices.

```
#include<stdio.h>
#include<conio.h>
int main()
{
    int matrix1[10][10],matrix2[10][10],matrix3[10][10],i,j,k,r1,c1,r2,c2;
    printf("Enter the size of Matrix1:\n");
    scanf("%d%d",&r1,&c1);
    printf("Enter the size of Matrix2:\n");
    scanf("%d%d",&r2,&c2);
```

```
if(c1==r2)
{
        printf("Enter the elements of Matrix1\n");
        for(i=0;i<r1;i++)
        {
                for(j=0;j<c1;j++)
                {       printf("Enter matrix1[%d][%d] element:",i,j);
                        scanf("%d",&matrix1[i][j]);
                }
        }
        printf("Enter the elements of Matrix2\n");
        for(i=0;i<r2;i++)
        {
                for(j=0;j<c2;j++)
                {
                        printf("Enter matrix2[%d][%d] element:",i,j);
                        scanf("%d",&matrix2[i][j]);
                }
        }
        printf("\nThe matrix1 is\n");
        for(i=0;i<r1;i++)
        {
                for(j=0;j<c1;j++)
                {
                        printf("%d\t",matrix1[i][j]);
                }
                printf("\n");
        }
        printf("\nThe matrix2 is\n");
        for(i=0;i<r2;i++)
        {
                for(j=0;j<c2;j++)
                {
                        printf("%d\t",matrix2[i][j]);
                }
                printf("\n");
        }
        for(i=0;i<r1;i++)
        {
                for(j=0;j<c2;j++)
                {
                        matrix3[i][j]=0;
                        for(k=0;k<c1;k++)
                        {
                                matrix3[i][j]+=matrix1[i][k]*matrix2[k][j];
                        }
                }
        }
        printf("\nThe Resultant Matrix is\n");
        for(i=0;i<r1;i++)
        {
                for(j=0;j<c2;j++)
                {
                        printf("%d\t",matrix3[i][j]);
                }
```

```
                        printf("\n");
                    }
                }//if close
                else
                    printf("Matrix Multiplicatin is not possible");
                getch();
                return 0;
            }
```

Output:



5.5 Strings

- The string is sequence of characters that is treated as a single data item. In general, the character array is called as string. Any group of characters defined between double quotation marks is a string constant. Such as –

  "The C Programming Language."
- Array of character type which is terminated by null characters is known as string.
- The terminating null character '\0' is important because it is the only way the string handling functions can know the string ends of string. Normally, each character is stored in one byte, and successive characters of the string are stored in successive bytes.
- In C, header file **string.h** provides special function for manipulating strings.

5.5.1 Initializing string:

A string is initialized as

char name[ ] = {'N', 'E', 'P', 'A','L'}

Then string name is initialized to NEPAL. This technique is also valid. But C offers special way to initialize the string as

char name[ ] = "NEPAL" ;

The characters of the string are enclosed within a pair of double quotes.

Example 5-7: Program to illustrate string initialization

```
#include<stdio.h>
#include<conio.h>
int main()
{
    char name[]="NEPAL ENGINEERING COLLEGE";
    printf("Name is %s",name);
    getch();
    return 0;
}
```

Output:

Name is NEPAL ENGINEERING COLLEGE

5.5.2 Arrays of Strings

- Arrays of string means two dimensional array of characters. For example

  char str[5] [8] ;

- The first dimension (size) tells how many strings are in the array. The second dimension tells the maximum length of each string. In above declaration, we can store 5 strings, each can store maximum 7 characters ; last : 8th space is for null terminator in each string.

❖ Initialization of array of string:

The array of string initialized like follow:

char str[2] [7] = {"Sunday", "Monday") ;

or like the following

```
char str[2] [ 7] = { { 's', 'u', 'n', 'd', 'a', 'y'} ,
                     { 'm', 'o', 'n', 'd', 'a', 'y'} } ;
```

❖ Accessing array of string

```
char a[4] [7] ;
```

If we want to read all the string from keyboard, we can use loop like follow:

```
for (i=0 ; i<=3 ; i++)
scanf("%d", a[i]) ;
```

If we want to print all the string then loop is

```
for (i=0 ; i<=3 ; i++)
printf ("%s", a[i]) ;
```

Example 5-8: Program to store name of week days and then print all

```
#include<stdio.h>
#include<conio.h>
void main( )
{
    int i;
    char day[7][10]={"sunday","monday","tuesday","wednesday","thursday","friday","saturday"};
    clrscr();
    printf("Weekdays are: \n");
    for (i=0;i<=6;i++)
    printf ("%s\n",day[i]);
    getch();
}
```

5.6 String Handling Function:

- The library or built-in functions strlen(), strcpy(), strcat(), strcmp(), strrev() etc are used for string manipulation. In order to use these function, we must include **string.h** file.

1. **strlen() function:**

   This function returns an integer which denotes the length of string passed. The length of string is the number of characters present in it, excluding the terminating null character. Its syntax is

   integer_variable = strlen(string) ;

Example 5-9:Program to find out the length of a string

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
int main()
{
```

```
        char name[40];
        int len;
        printf("Enter your name :");
        gets(name);
        len = strlen(name);
        printf ("\n The number of character in your name is :%d",len);
        getch();
        return 0;
    }
```

Output:

```
Enter your name :Naresh Prasad Das
The number of character in your name is :17
```

2. strcpy() function

The strcpy( ) function copies one string to another. The function accepts two string as parameters and copies the second string character by character into the first one upto including the null character of the second string. The syntax is

strcpy(destination_string, source_string) ;

i.e. strcpy(s1, s2) ; means the content of s2 is copied to s1.

Example5-10: Program to copy one string to another

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
int main( )
{
    char name[]="Nepal Engineering College",s[20];
    strcpy(s,name) ;
    printf("The copied string is %s",s);
    getch();
    return 0;
}
```

Output:

```
The copied string is Nepal Engineering College
```

3. strcat() fuction

This function concatenates two strings i.e. it appends one string at the end of another. This function accepts two strings as parameters and stores the contents of the second string at the end of the first. Its syntax is:

strcat (string1, string2);

i.e. string1 = string1 + string2 ;

Example 5-11: Program to concate two strings

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
int main( )
{
    char str1[20]="Hello",str2[]="World";
    strcat(str1,str2);
    printf("\n%s",str1);
    getch();
    return 0;
}
```

Output:

```
HelloWorld
```

4. strcmp() function

- This function compares two strings to find out whether they are same or different. This function accepts two string as parameters and returns an integer whose value is
  - i. less than 0 if the first string is less than the second
  - ii. equal to 0 if both are same
  - iii. greater than 0 if the first string is greater than the second
- Two strings are compared character by character until there is a mismatch or end of one of strings is reached. Whenever two characters in two strings differ, the string which has the character with a higher ASCII value is greater. For example, consider two string "ram" and "rakesh". The first two character are same but third character in string ram and that is in rakesh are different. Since ASCII value of character m in string ram is greater than that of k in string rakesh, the string ram is greater than rakesh.
- Its syntax

integer_variable = strcmp(string1, string2) ;

Example 5-12:Program to illustrate the use of strcmp( )

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
int main( )
{       char str1[15],str2[15];
        int diff;
        printf ("\n Enter first string:");
        gets(str1);
        printf ("\n Enter second string:");
        gets(str2);
        diff=strcmp(str1,str2);
        if(diff==0)
                printf("Both String are same");
        else if(diff>0)
                printf("%s is greater than %s",str1,str2);
        else
         printf("%s is smaller than %s",str1,str2);
         getch();
         return 0;
}
```

Output:

```
Enter first string:ram
Enter second string:rakesh
ram is greater than rakesh
```

5. strupr() function

This function converts the lower case string into upper case.

Example 5-13:

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
int main()
{
        char str1[20]="hello";
        strupr(str1);
        puts(str1);
         getch();
         return 0;
}
```

Output: HELLO

6. strlwr() function

This function converts the upper case string to lower case.

Example 5-14:

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
int main()
{
        char str1[20]="HELLO";
        strlwr(str1);
        puts(str1);
         getch();
         return 0;
}
```

Output: hello

7. strrev() function

This function is used to reverse all characters in a string except null character at the end of string. The reverse of string "abc" is "cba".

Its syntax is: **strrev(string) ;**

For example:

strrev(s) means it reverses the characters in string s and stores reversed string in s.

Example 5-15:

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
int main()
{
        char str1[20]="hello";
        strrev(str1);
        puts(str1);
         getch();
         return 0;
}
```

Output: olleh

Example 5-16: Write a program to check whether a given word is *palindrome* or not.

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
#include<stdlib.h>
main()
{
    char str[30];
    int l,i;
    printf("Enter a word:");
    gets(str);
    l=strlen(str);
    for(i=0;i<l/2;i++)
    {
        if(str[i]!=str[l-i-1])
        {
            printf("It is not palindrome");
            getch();
            exit(1);
        }
    }
    printf("It is palindrome");
    getch();
}
```

Output:

Enter a word:madam
It is palindrome

Some Solved Example

1. **Write a C Program to insert a character in any desired location in a string.**
   For example if original string is "NEPL" and 'A' is to be inserted at index position 3; program should output "NEPAL".

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
int main()
{
        char str[20],ch;
        int l,i,p;
        puts("Enter string: ");
        gets(str);
        puts("Enter index position where you want to insert:");
        scanf("%d",&p);
        puts("Enter a character which you want to insert:");
        fflush(stdin);
        scanf("%c",&ch);
        l=strlen(str);
        for(i=l+1;i>p;i--)
        {
                str[i]=str[i-1];
        }
        str[p]=ch;
        puts("Final string is:");
        puts(str);
        getch();
        return 0;
}
```

2. **Print the following pattern using a C program. Take the string as an input from the user. Write a generic program to accept string of any length.**

   Example pattern

   If the user input is NEPAL than the output should be as follows:

```
1 N
2 E E
3 P P P
4 A A A A
5 L L L L L
```

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
int main()
{
        char str[50];
        int i,j,l;
        printf("Enter a string:");
        gets(str);
        l=strlen(str);
        for(i=0;i<l;i++)
        {
                printf("%d ",i+1);
                for(j=0;j<=i;j++)
```

```
            {
                printf("%c ",str[i]);
            }
            printf("\n");
        }
        getch();
        return 0;
    }
```

3. **Write a program to convert all the uppercase letter to lowercase and vice-versa in a string given by the user.**

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
#include<ctype.h>
int main()
{
    char str[50];
    int i,l;
    printf("Enter string:");
    gets(str);
    l=strlen(str);
    for(i=0;i<l;i++)
    {
        if(islower(str[i]))
            str[i]=toupper(str[i]);
        else
            str[i]=tolower(str[i]);
    }
    puts(str);
    getch();
    return 0;
}
```

Output:

```
Enter string:tHE c pRogramming
The C PrOGRAMMING
```

4. **Write a C program to print all the locations at which required element is found and also the number of times it occur in the list. [Linear search for multiple occurrences]**

```
#include<stdio.h>
#include<conio.h>
int main()
{
    int array[100],search,i,n,count=0;
    printf("Enter the number of elements in array\n");
    scanf("%d", &n);
    printf("Enter %d numbers\n", n);
    for (i=0;i<n;i++)
        scanf("%d",&array[i]);
    printf("Enter the number to search\n");
        scanf("%d",&search);
    for (i=0;i<n;i++)
    {
        if (array[i] == search)
        {
```

```
                printf("%d is present at location %d.\n", search,i+1);
                count++;
            }
        }
        if (count == 0)
            printf("%d is not present in array.\n", search);
        else
            printf("%d is present %d times in array.", search, count);
        getch();
        return 0;
    }
```

Output:

```
Enter the number of elements in array
5
Enter 5 numbers
8
5
2
5
3
Enter the number to search
5
5 is present at location 2.
5 is present at location 4.
5 is present 2 times in array.
```

5. **Write a program to declare names of 10 persons in an array in the program already and input any name from keyboard and check it is present in name list or not.**

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
#include<stdlib.h>
int main()
{
  char list[10][20]={"Ram","Hari","sita","ganesh","narayan","anil","bikram","pravin","prakrit","pritam"};
  char name[20];
  int i;
  printf("Enter name you want to search:");
  gets(name);
  for(i=0;i<10;i++)
  {
        if(strcmpi(list[i],name)==0)
        {
            printf("%s is present in the list",name);
            getch();
            exit(1);
        }
  }
  printf("%s is not present in the list",name);
  getch();
  return 0;
}
```

6. **Write a program to scan names of different persons and arrange them in alphabetic order.**

```
#include <stdio.h>
#include <conio.h>
#include <string.h>
int main()
{
   char name[20][40],temp[50];
```

```
    int i,j,n;
    printf("Enter number of person:");
    scanf("%d",&n);
    for(i=0;i<n;i++)
    {
       printf("\n Enter the name %d :- ",i+1);
       scanf("%s",name[i]);
    }
    for(i=0;i<n-1;i++)
    {
            for(j=0;j<n-i-1;j++)
            {
                    if(strcmp(name[j],name[j+1])>0)
                    {
                            strcpy(temp,name[j]);
                            strcpy(name[j],name[j+1]);
                            strcpy(name[j+1],temp);
                    }
            }
    }
    printf("Name in alphabetic order:");
    for(i=0;i<n;i++)
    {
       printf("\n%s",name[i]);
    }
    getch();
   return 0;

}
```

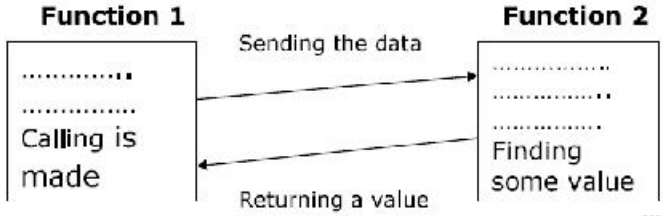
Output:

```
Enter number of person:3

 Enter the name 1 :- ram

 Enter the name 2 :- rakesh

 Enter the name 3 :- hari
Name in alphabetic order:
hari
rakesh
ram
```

6 Functions

6.1 Introduction

- A function is a self-contained program segment that carries out some specific, well-defined task.
- This is a logical unit composed of a number of statements grouped into a single unit. It can also be defined as a section of a program performing a specific task.
- Every C programs consist of one or more programs. Execution of program will always begin by carrying out the instructions in main. Additional function will be subroutine to main.
- Functions are used to encapsulate a set of operations and return information to the main program or calling routine.



- **Function1** is the calling function and **Function2** is called function
- Example

```
#include<stdio.h>
#include<conio.h>
int main()
{
        Printf("Welcome to the Programming World\n");
        getch();
        return 0;
}
```

- This program prints the message "Welcome to the Programming World" at the output terminal. The same program can be written using function concept as

```
#include<stdio.h>
void PrintMessage()
{
        Printf("Welcome to the Programming World");
}
int main()
{
        PrintMessage();
        getch();
        return 0;
}
```

Benefits of Using Functions

- It makes program significantly easier to understand and maintain.
- Well written functions may be reused in multiple programs. The C standard library is an example of the reuse of functions.
- Different programmers working on one large project can divide the workload by writing different functions
- Program that uses functions are easier to design, program, debug and maintain.

6.2 Types of Functions

- C program has two types of functions:
    1. Library Functions
    2. User defined functions

1. Library Functions:

These are the functions which are already written, compiled and placed in C Library and they are not required to be written by a programmer. The function's name, its return type, their argument number and types have been already defined. We can use these functions as required. For example: printf(), scanf(), sqrt(), getch(), etc.

2. User defined functions

- Functions defined by the users according to their requirements are called user-defined functions.
- These are the functions which are defined by user at the time of writing a program. The user has choice to choose its name, return type, arguments and their types. The job of each user defined function is as defined by the user. A complex C program can be divided into a number of user defined functions.

What is about main( ) function?

The function main() is an user defined function except that the name of function is defined or fixed by the language. The return type, argument and body of the function are defined by the programmer as required. This function is executed first, when the program starts execution.

❖ Elements of user-defined functions

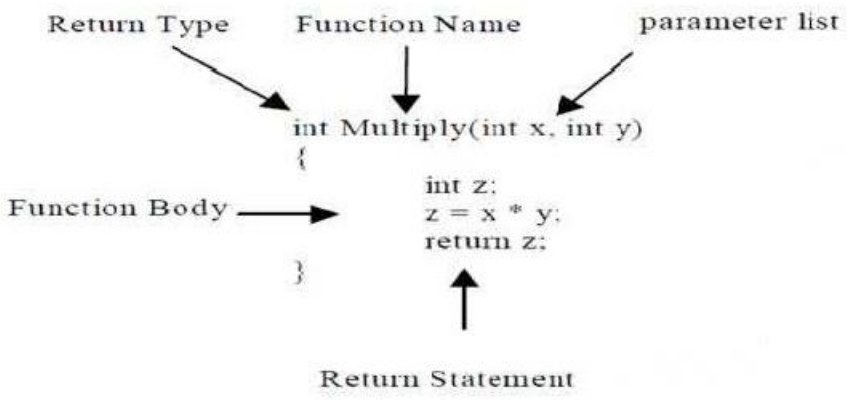
- There are some similarities exists between functions and variables in C.
  - Both function name and variable names are considered as identifiers and therefore they must follow the rules of creating variable's name.
  - Like variables, functions have data types associated with them.
  - Like variables, function names and their types must be declared and defined before they are used in the program.
- In order to make use of user defined functions, we need to establish three elements that are related to functions.
  1. Function definition
  2. Function call
  3. Function declaration

6.2.1 Definition of functions

- The function definition is an independent program module that is specially written to implement the requirements of the function. So it is also called as function implementation. It includes following topics:
  - Function Header
  - Function Body
- The general form of function definition is as given below:

```
function_type function_name(parameters list)
{
        local variables declaration;
        executable statement1;
        executable statement2;
        ------;
        ------;
        return statement;
}
```

- The first line function_type function_name(parameters list) is known as function header and the statements enclosing the curly braces are known as function body.
- Example



❖ Function Header

- It includes three parts:
  1. **Function type**
     It specifies type of the value that the function is expected to return to the program calling the function. It is also called as return type. If function is not returning any values we have to specify it as void.
  2. **Function name**
     It is any valid C identifier name and therefore must follow the rules for creation of variable names in C.
  3. **Parameters list**
     It declares variables that will receive the data sent by the calling program. They serve as input data to the function to carry out specific task. Since, they represent the actual input values they are referred as formal parameters or formal arguments.
- For Example

```
int findmax(int x, int y int x)
 {
   --------
 }
double power(double x, int n)
 {
   --------
 }
float quad(int a, int b, int c)
 {
   --------
 }
void print_line()
 {
   --------
 }
```

❖ Function Body

- It contains the declarations and statements necessary for performing required task. The body enclosed in braces, contains three parts:
  1. **Local variables declaration**
     It specifies the variables needed by the function locally.
  2. **Function Statements**
     That actually performs task of the function.
  3. **The return statement**
     It returns the value specified by the function.
- **For Example**

```
float mul(float x, float y)
{
 float result;                /* local variable */
   result = x * y;            /* find the result */
   return(result);            /* return the result */
}
void display(void)
{
   printf("Hello World!"); /* only print the value */
}
void sub(int a, int b)
{
 printf("Subtraction: %d", a – b); /* no variables */
   return;                         /* optional */
}
```

❖ **Return values**

- A function may or may not send back any value to the calling function. If it does, it is done by return statement. The return statement also sends the program control back to the calling function. It is possible to pass a calling function any number of values but called function can only return one value per call, at most.
  The return statement can take one of the following forms:

```
return;
return(value);
return(variable);
return(expression);
```

- The first plain return does not return any value; it acts much as closing brace of the function. The remaining three statements can eliminates the brackets also. When the return is executed, the program control immediately transfers back to the calling function. None of the statements written after return are executed afterwards. For example:

```
return(x);
printf("Bye…Bye");
```

- In this case the printf statement will not be executed in any case. Because the return statement will transfer program control immediately back to the calling function.
- Some examples of return are:

```
int div(int x, int y)
{
    int z;
    z = x / y;
    return z;
}
```

Here, instead of writing three statements inside the function div(), we can write a single statement as,

```
return(x/y);
OR
return x/y;
```

A function may have more than one return statement when it is associated with any condition such as,

```
if(x>y)
        return x;
else
        return y;
```

In this code, in any condition, only one return statement will be executed.

6.2.2 Function Calls

- A function can be called by simply using function name followed by a list of actual parameters (or arguments) if any, enclosed in parentheses. For example:

```
main()
{
        float m;
        m = mul(5.2, 3.71);        /* function call */
        printf("\n%d",m);
}
```

- When compiler encounters the function call, it transfers program control to the function mul() by passing values 5.2 and 3.71 (actual parameters) to the formal parameters on the function mul(). The mul() will performs operations on these values and then returns the result. It will be stored in variable m. We are printing the value of variable m on the screen. There are a number of different ways in which the function can be called as given below:

```
mul(5.2, 3.71);
mul(m, 3.71);
mul(5.2, m);
mul(m, n);
mul(m+5.1, 3.71);
mul(m+3.6, m+9.1);
mul(mul(6.1,8.2), 4.5);
```

- Only we have to remember that the function call must satisfy type and number of arguments passed as parameters to the called function.

6.2.3 Function Prototype / Function Declaration

- Like variables, all functions in C program must be declared, before they are used in calling function. This declaration of function is known as function prototype. It is having following syntax to use:

  function_type function_name (parameter list);

- This is very similar to function header except the terminating semicolon. For example, the function mul() will be declared as,

  float mul(float, float);

- The function with the prototype double power(double x, int n); takes one double and one int data and returns a double type data.

6.3 Types of user defined function

- The functions can be classified into four categories on the basis of the arguments and return values:

  4. Functions with no arguments and no return value
  5. Functions with no arguments and return value
  6. functions with arguments and no return value
  7. Functions with arguments and return value

6.3.1 Functions with no arguments and no return value

When a function has no arguments, it does not receive any data from the calling function. Similarly, when called function does not return a value, the calling function does not receive any data from it. Thus, in such type of functions, there is no data to transfer between the calling function and the called function. This type of function is defined as

```
void function_name( )
{
        /* body of function */
}
```

The keyword void means the function does not return any value. There is no argument within parenthesis which implies function has no argument and it doesn't receive any data from the called function.

Example 6-1:

```
#include<stdio.h>
#include<conio.h>
 void add( )
 {
        int a,b,sum;
        printf ("Enter two numbers: ");
        scanf ("%d%d", &a, &b);
        sum = a + b;
        printf ("\nThe sum is %d", sum);
 }
 main( )
 {
         add( );
         getch();
 }
```

Output:

```
Enter two numbers: 5   4
The sum is 9
```

6.3.2 Functions with no arguments and return value

These type of functions do not receive any arguments but they can return a value.

```
int func();
main()
{
        int r;
        r=func();
        ……
}
 int func()
{
        …….
        return (expressions) ;
}
```

Example 6-2:

```
#include<stdio.h>
#include<conio.h>
int add();
main()
        {
                int r;
                r=add();
                printf (" The sum is %d\n",r);
                getch();
        }
int add()
{
        int n1,n2,sum ;
        printf ("Enter two numbers:");
        scanf ("%d%d",&n1,&n2);
        sum = n1+n2;
        return (sum);
}
```

Output:

```
Enter two numbers: 5   4
The sum is 9
```

6.3.3 Functions with arguments and no return value

These types of functions have arguments, hence the calling function can send data to the called function but the called function does not return any value. These functions can be written as

```
void func (int, int) ;
main( )
{
        …….
        func(a, b) ;
        …….
}
void func( int c, int d)
{
        ……..
        statements;
        …….
}
```

Example 6-3:

```
#include<stdio.h>
#include<conio.h>
void add(int,int);
main()
{
        int a,b;
        printf ("Enter two numbers:");
        scanf ("%d%d",&a,&b);
        add (a,b);
        getch();
}
void add(int c,int d)
{
        int sum;
        sum = c+d;
        printf ("\nThe sum is %d",sum);
}
```

Output:

```
Enter two numbers: 5   4
The sum is 9
```

6.3.4 Functions with arguments and return value

These types of functions have arguments, so the calling function can send data to the called function, it can also return any values to the calling function using return statement. This function can be written as

```
int func (int, int);
main( )
{
        int r;
        ……..
        r = func(a,b);
        ……..
}
int func(int a, int b)
{
        ……..
        return(expression);
}
```

Example 6-4:

```
#include<stdio.h>
#include<conio.h>
int add(int,int);
main()
{
        int a,b,sum;
        printf ("Enter two numbers:");
        scanf ("%d%d",&a,&b);
        sum = add(a,b);
        printf ("\nthe sum is %d",sum);
        getch();
}
```

```
int add(int a,int b)
{
        int sum;
        sum = a+b;
        return sum;
}
```

Output:

```
Enter two numbers: 5   4
The sum is 9
```

Example 6-5:

```
#include<stdio.h>
#include<conio.h>
int findmax(int,int);
main()
{
        int a,b,r;
        printf ("Enter two numbers:");
        scanf ("%d%d",&a,&b);
        r = findmax(a,b);
        printf ("\nLargest number is %d",r);
        getch();
}
int findmax(int c,int d)
{
        if(c>d)
                return c;
        else
                return d;
}
```

Output:

```
Enter two numbers: 5   4
Largest number is 5
```

6.4 Functions Parameters:

- An argument is an entity used to pass the data from calling function to the called function. The argument are also called as the parameters.
- Parameters provide the data communication between calling function and called function.
- *Two types:*
  - **Actual parameters:**
    - These are the parameters transferred from the calling function (main function) to the called function (user defined function).
    - This is the argument which is used in function call.
  - **Formal parameters:**
    - Passing the parameters from the called functions (user defined function) to the calling function (main function).
    - This is the argument which is used in function definition.

6.5 Passing argument to function

- Functions communicate with each other by passing the arguments. There are two ways of passing the arguments namely, call by value and call by reference.

6.5.1 Call by value/ Pass by value

- In this *call by value* method the value of each of the actual arguments in the calling function is copied into corresponding formal arguments of the called function.
- With this method the changes made to the formal arguments in the called function have no effect on the values of actual argument in the calling function.

Example 6-6:

```
#include<stdio.h>
#include<conio.h>
int swap(int x,int y);
main()
{
        int a,b,r;
        printf ("Enter two numbers:");
        scanf ("%d%d",&a,&b);
        printf("\nValue before swap a=%d and b=%d",a,b);
        swap(a,b);
        printf("\n\nValue inside main function");
        printf("\nValue after swap a=%d and b=%d",a,b);
        getch();
}
int swap(int x,int y)
{
        int t;
        t=x;
        x=y;
        y=t;
        printf("\n\nValue inside swap function");
        printf("\nValue after swap a=%d and b=%d",x,y);
}
```

Output:

```
Enter two numbers:5 6

Value before swap a=5 and b=6

Value inside swap function
Value after swap a=6 and b=5

Value inside main function
Value after swap a=5 and b=6
```

6.5.2 Call by Reference/ Pass by Reference

- In the *call by reference* method the address of actual arguments in the calling function are copied into formal arguments of the called function. This means that using these addresses the actual arguments can be accessed.
- In call by reference since the address of the value is passed any changes made to the value reflects in the calling function.

Example 6-7: Write a program to swap two numbers passing reference as argument.

```
#include<stdio.h>
#include<conio.h>
int swap(int *x,int *y);
main()
{
        int a,b,r;
        printf ("Enter two numbers:");
        scanf ("%d%d",&a,&b);
        printf("\nValue before swap a=%d and b=%d",a,b);
        swap(&a,&b);
        printf("\n\nValue inside main function");
        printf("\nValue after swap a=%d and b=%d",a,b);
        getch();
}
int swap(int *x,int *y)
{
       int t;
       t=*x;
       *x=*y;
       *y=t;
       printf("\n\nValue inside swap function");
       printf("\nValue after swap a=%d and b=%d",*x,*y);
}
```

Output:

```
Enter two numbers:5 6

Value before swap a=5 and b=6

Value inside swap function
Value after swap a=6 and b=5

Value inside main function
Value after swap a=6 and b=5
```

6.6 Local and global variable

Local Variable:

- The variables that are defined within the body of a function or a block, are local to that function or block only and are called local variables. The local variables are created when the function is called and destroyed automatically when the function is exited.
- Its name and value are valid within the function in which it is declared. They are unknown to other functions.

Example 6-8:

```
#include<stdio.h>
#include<conio.h>
void test();
int main()
{
```

```
        int x=5;
        printf("The value of x inside main function is %d",x);
        test();
        getch();
        return 0;
}
void test()
{
        int x=10;
        float y=5.5;
        printf("\nThe value of x and y inside test function is x=%d and y=%f",x,y);
}
```

Output:

The value of x inside main function is 5
The value of x and y inside test function is x=10 and y=5.500000

- Here each x and y variables are LOCAL to its own routine and are called local variables. The x variables in main() is unrelated to the x variable in the test function.

Global Variable:

- The variables that are defined outside any function are called global variables. All functions in the program can access and modify global variables.
- The global variable is accessible to all the function defined in the program.
- It is useful to declare a variable global if it is to be used by many functions in the program. The default initial value for these variable is zero. The scope is global i.e. within the program. The life time is as long as the program's execution does not come to an end.

Example 6-9:

```
#include<stdio.h>
#include<conio.h>
void test();
int x;
float y=4.5;
int main()
{
        x++;
        printf("The value of x and y inside main function is x=%d and y=%f",x,y);
        test();
        getch();
        return 0;
}
void test()
{
        x++;
        printf("\nThe value of x and y inside test function is x=%d and y=%f",x,y);
}
```

Output:

The value of x and y inside main function is x=1 and y=4.500000
The value of x and y inside test function is x=2 and y=4.500000

6.7 Storage Classes

- Storage class is a concept in C which provides information about the variable's visibility, lifetime and scope.
  Scope: Scope is the region in which a variable is available for use.
  Visibility: The program's ability to access a variable from memory is called as visibility of variable.
  Lifetime: The lifetime of the variable is duration of the time in which a variable exists in the memory during execution.
- The properties of the variables tell us about following things, referred as storage class:
  - Where the variable would be stored?
  - What will be the initial value of the variable? (i.e. default initial value)
  - What is scope of the variable?
  - What is lifetime of the variable?
- Apart from data type variable have storage class. In C, there are four types of storage classes. They are
  1. Local or Automatic Variables
  2. Global or External Variables
  3. Static Variables
  4. Register Variables

6.7.1 Local or Automatic Variables

- Automatic variables are declared inside a particular function and they are created when the function is called and destroyed when the function exits.
- Automatic variables are local or private to a function in which they are defined. Other names of automatic variable are internal variable and local variable.
- A variable which is declared inside the function without using any storage class is assumed as the automatic variable because the default storage class is automatic
  Storage: Memory
  Default initial Value: Garbage
  Scope: Local (to block in which variable defined)
  Lifetime: Till control remains within block where it is defined
  Syntax
  auto<variable>
- Here auto is the keyword for automatic variable

Example 6-10:

```
main()
{
        auto int x,y;
        int a,b;
        x=10;
        printf("Values: x=%d and y=%d",x,y);
}
```

- Here x and y are defined as automatic variables by the usage of keyword auto. Variables a and b are also considered as auto because any variable which does not have any storage class explicitly specified, it is considered as automatic storage class.

6.7.2 Global or External Variables

- External variable is a global variable which is declared outside the function. The memory cell is created at the time of declaration statement is executed and is not destroyed when the flow comes out of the function to go to some other function.
- The global variable can be accessed by any function in the same program. A global variable can be declared externally in the global variable declaration section.
  Storage: Memory
  Default initial value: Zero
  Scope: Global
  Lifetime: As long as the program execution doesn't come to the end

Syntax:

There is no keyword to state any variable as global. If any variable is declared outside the function without any other storage class, it is implicitly considered as global variable

Example 6-11:

```
#include<stdio.h>
#include<conio.h>
void f1();
void f2();
int x=100;
main()
{
        x=200;
        f1();
        f2();
        getch();
}
void f1()
{
        x=x+1;
}
void f2()
{
        x=x+100;
        printf("The final value of global variable x is %d",x);
}
```

Output:

The final value of global variable x is 301

- Here global variable x is declared above all the function. It can be accessed by all the function as main(), f2() and f2(). The initial value assigned to it is 100. When it is accessed by main. Its value is 200. When f1() function accessed it, x with the last recent value 200 is passed and f1() function manipulates it to 201. Similarly when function f2() is executed the value of x becomes 301 which is given in output

6.7.3 Static Variables

- These variables are alive throughout the program. A static variable can be initialized only once at the time of declaration. The initialization part is executed only once and retains the value till the end of the program.

Storage: Memory
Default initial Value: Zero
Scope: Local (to block in which variable defined)
Lifetime: Value of the variable remains as it is between function calls

Syntax:

A variable can be declared as static by using the keyword "static"

Example 5-12:

```
#include<stdio.h>
#include<conio.h>
void stat();
main()
{
        int j;
        for(j=0;j<3;j++)
```

```
                stat();
        getch();
}
void stat()
{
        static int x;
        x=x+1;
        printf("\nValue of x is %d",x);
}
```

Output:

```
Value of x is 1
Value of x is 2
Value of x is 3
```

- During the first call to stat() function, x is incremented to 1. Because x is static, this value persists and therefore the next call adds another 1 to x giving it a value of 2. The value of x becomes 3 when third call is made. If variable x would have declared as an auto then output would have been x=1 all the three times.

6.7.4 Register Variables

- A variable is usually stored in the memory but it is also possible to store a variable in the processor's register by defining it as register variable. The registers access is much faster than a memory access, keeping the frequently accessed variables in the register will make the execution of the program faster. Since only a few variables can be places in a register, it is important to carefully select the variables for this purpose. However C will automatically convert register variables into normal variables once the limit exceeded.

  Storage: CPU Register
  Default initial Value: Garbage
  Scope: Local (to block in which variable defined)
  Lifetime: Till control remains within block where it is defined
  Syntax
  Register variable are declared by using the keyword "register"

Example 6-13:

```
#include<stdio.h>
#include<conio.h>
main()
{
        register int x,y,z;
        printf("Enter two numbers:");
        scanf("%d%d",&x,&y);
        z=x+y;
        printf("\nThe Sum is %d",z);
        getch();
}
```

Output:

```
Enter two numbers:5     6
The Sum is 11
```

- In the above program, all the variable are stores in the registers instead of memory.

6.8 Pre-Processor Directives

- Preprocessor directives are lines included in the code of programs preceded by a hash sign (#). These lines are not program statements but directives for the preprocessor. The preprocessor examines the code before actual compilation of code begins and resolves all these directives before any code is actually generated by regular statement.
- These preprocessor directives extend only across a single line of code. As soon as a newline character is found, the preprocessor directive is ends. No semicolon (;) is expected at the end of a preprocessor directive.
-

Directive	Description
#define	Substitutes a preprocessor macro.
#include	Inserts a particular header from another file.
#undef	Undefines a preprocessor macro.
#ifdef	Returns true if this macro is defined.
#ifndef	Returns true if this macro is not defined.
#if	Tests if a compile time condition is true.
#else	The alternative for #if.
#elif	#else and #if in one statement.
#endif	Ends preprocessor conditional.
#error	Prints error message on stderr.

- Example
  #include<stdio.h>
  #define PI 3.14156

6.9 Macros

- A macro is a fragment of code which has been given a name. Whenever the name is used, it is replaced by the contents of the macro.
- **Syntax**
  #define macro_name replacement_text
- Example:
  #define PI 3.14156

Example 6-14:

```
#include<stdio.h>
#include<conio.h>
#define max(a,b) ((a>b)?(a):(b))
int main()
{
        int a,b;
        printf("Enter two numbers:");
        scanf("%d%d",&a,&b);
        printf("Maximum is %d",max(a,b));
        getch();
        return 0;
}
```

Output:

```
Enter two numbers:5 6
Maximum is 6
```

6.10 Header files

- A header file is a file with extension .h which contains C function declarations and macro definitions to be shared between several source files. There are two types of header files: the files that the programmer writes and the files that comes with your compiler.
- Including a header file is equal to copying the content of the header file but we do not do it because it will be error-prone and it is not a good idea to copy the content of a header file in the source files, especially if we have multiple source files in a program.
- Both the user and the system header files are included using the preprocessing directive #include. It has the following two forms:

#include <file>	This form is used for system header files.
#include "file"	This form is used for header files of your own program.

6.11 Recursive Functions

Recursion in programming is a technique for defining a problem in terms of one or more smaller versions of the same problem. The solution of the problem is built on the results from the smaller versions. A recursive function is one that calls itself directly or indirectly to solve a smaller version of its task until a final call which does not require a self-call. Thus, the function is called recursive function if it calls to itself and recursion is a process by which a function call itself repeatedly until some specified condition will be satisfied. This process is used for repetitive computations in which each action is stated in term of previous result. Many iterative or repetitive problems can be written in this form.

- To solve a problem using recursive method, two conditions must be satisfied. They are:
  - Problem could be written or defined in terms of its previous result.
  - Problem statement must include a stopping condition i.e. we must have an if statement somewhere to force the function to return without the recursive call being executed, otherwise function will never return.

Example 6-15: WAP to find the factorial of a number using recursive method

```
#include<stdio.h>
#include<conio.h>
long int factorial(int n)
{
        if (n==1)
                return(1);
        else
                return(n*factorial(n-1));
}
int main()
{
        int num;
        printf("Enter a number:");
        scanf ("%d",&num);
        printf("The factorial is %ld",factorial(num));
        getch();
        return 0;
}
```

Output:

```
Enter a number:5

The factorial is 120
```

Example 6-16: Fibonacci using recursion

```
#include<stdio.h>
#include<conio.h>
int fibonacci(int);
int main()
{
        int n,i,r;
        printf("Enter numbers of term:");
        scanf("%d",&n);
        printf("Fibonacci series:\n");
        for(i=0;i<n;i++)
        {
                r=fibonacci(i);
                printf("%d\t",r);
        }
        getch();
        return 0;
}
int fibonacci(int n)
{
        if(n==0||n==1)
                return n;
        else
        return(fibonacci(n-1)+fibonacci(n-2));
}
```

Output:

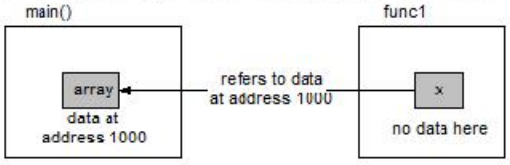
```
Enter numbers of term:10
Fibonacci series:
0       1       1       2       3       5       8       13      21      34
```

6.12 One dimensional arrays and functions

- In C, arrays are automatically passed by reference to a function. The name of an array stores the beginning address of where the array data is stored in memory. When we pass an array by reference, the subroutine (function) has access to all the elements stored in the array and any change in the function will be reflected to the calling function.
- An array can be passed to a function by passing only the array-name and size of the array.
- When we pass array by value, the function has a physically separate local copy. The function can access and even modify the array elements without effecting in the calling function.
- Example

```
main()
{
    int array[20];
    func1(array);          /* passes pointer to array to func1 */
}
```

Since we are passing the address of the array the function will be able to manipulate the actual data of the array in main(). This is call by reference as we are not making a copy of the data but are instead passing its address to the function. Thus the called function is manipulating the same data space as the calling function.



- In the function receiving the array the formal parameters can be declared in one of three almost equivalent ways as follows:
  - As a sized array :

```
func1 ( int x[10] )
{
        ...
}
```

  - As an unsized array :

```
func1 ( int x[ ] )
{
        ...
}
```

  - As an actual pointer

```
func1 ( int *x )
{
        ...
}
```

All three methods are identical because each tells us that in this case the address of an array of integers is to be expected.

Example 6-17: Write a program to sort n elements of an array using user defined function.

```
#include<stdio.h>
#include<conio.h>
void sort_array(int[], int n);
void display_array(int[], int n);
main()
{
        int marks[50],i,n;
        printf("Enter number of element:");
        scanf("%d",&n);
        for(i=0;i<n;i++)
        {
                printf("Enter element marks[%d]:",i);
                scanf("%d",&marks[i]);
        }
        sort_array(marks,n);
        display_array(marks,n);
        getch();
}
void sort_array(int marks[],int n)
{
        int i,j,temp;
        for(i=0;i<n;i++)
        {
                for(j=i+1;j<n;j++)
                {
                        if(marks[i]>marks[j])
                        {
                                temp=marks[i];
                                marks[i]=marks[j];
                                marks[j]=temp;
                        }
                }
        }
}
```

```
void display_array(int m[],int n)
{
        int i;
        printf("Sorted array is:\n");
        for(i=0;i<n;i++)
        {
                printf("%d\t",m[i]);
        }
}
```

Output:

```
Enter number of element:5
Enter element marks[0]:55
Enter element marks[1]:45
Enter element marks[2]:65
Enter element marks[3]:75
Enter element marks[4]:60
Sorted array is:
45      55      60      65      75
```

6.13 Passing two-dimensional arrays to function

Just as in one-dimensional array, when a two-dimensional array is passed as a parameter, the base address of the actual array is sent to the function (pass by reference). Any change made to the element of an array element inside a function will carry over to the original location of the array that is passed to the function. The size of all dimensions except the first must be included in the function heading and prototype. The size of those dimensions for formal parameters must be exactly the same as in the actual array. The function header and prototype specify the number of columns as a constant.

- For example, the following function declarations are valid

```
void function1(int x[8][5], int cs[]);
int function2(int x[][5],float m);
int function3(int[][5],float);
```

Example 6-18: Write a Program to add two 2 x 2 matrices using user defined function.

```
#include<stdio.h>
#include<conio.h>
void mat_read(int mat[2][2]);
void mat_print(int mat[2][2]);
void mat_add(int mat1[][2],int mat2[][2],int mat3[][2]);
main()
{
        int mat_a[2][2],mat_b[2][2],mat_res[2][2];
        puts("Enter First Matrix:-\n");
        mat_read(mat_a);
        puts("\nMatrix a is :-\n");
        mat_print(mat_a);
        puts("Enter Second Matrix:-\n");
        mat_read(mat_b);
        puts("\nMatrix b is :-\n");
        mat_print(mat_b);
        mat_add(mat_a,mat_b,mat_res);
        puts("The resultant matrix is\n");
        mat_print(mat_res);
        getch();
}
```

```
void mat_read(int mat[2][2])
{
        int j,k;
        for (j=0;j<2;j++)
        {
                for (k=0;k<2;k++)
                {
                        printf("Enter mat[%d][%d]:",j,k);
                        scanf("%d",&mat[j][k]);
                }
        }
}
void mat_print(int mat[2][2])
{
        int j,k;
        for (j=0;j<2;j++)
        {
                for (k=0;k<2;k++)
                {
                        printf("%d\t",mat[j][k]);
                }
                printf("\n");
        }
}
void mat_add(int mat1[][2],int mat2[][2],int mat3[][2])
{
        int j,k;
        for (j=0;j<2;j++)
                for (k=0;k<2;k++)
                        mat3[j][k] = mat1[j][k]+mat2[j][k];
}
```

Output:

```
Enter First Matrix:-

Enter mat[0][0]:1
Enter mat[0][1]:2
Enter mat[1][0]:3
Enter mat[1][1]:4

Matrix a is :-

1       2
3       4
Enter Second Matrix:-

Enter mat[0][0]:5
Enter mat[0][1]:6
Enter mat[1][0]:7
Enter mat[1][1]:8

Matrix b is :-

5       6
7       8
The resultant matrix is

6       8
10      12
```

Some Solved Example

1. **Write a function that takes an integer number of any digits and return the number revered. In the calling function read a number, pass it to the function, and display the result.**

```
#include<stdio.h>
#include<conio.h>
int rev(int);
int main()
{
        int n;
        printf("Enter a integer number:");
        scanf("%d",&n);
        printf("Reversed number=%d",rev(n));
        getch();
        return 0;
}
int rev(int num)
{
        int reverse=0,rem;
        while(num!=0)
        {
                rem=num%10;
                reverse=reverse*10+rem;
                num=num/10;
        }
        return reverse;
}
```

Output:

```
Enter a integer number:12653
Reversed number=35621
```

2. **Write a program to read an integer number, count the even digits and odd digits present in the entered number. You must write a function for calculating the result and the result must be displayed in main function itself.**

```
#include<stdio.h>
#include<conio.h>
void odd_even_digit_count(int,int*,int*);
int main()
{
        int n,odd_count=0,even_count=0;
        printf("Enter an integer number:");
        scanf("%d",&n);
        odd_even_digit_count(n,&odd_count,&even_count);
        printf("Numbers of odd numbers=%d",odd_count);
        printf("\nNumber of even numbers=%d",even_count);
        getch();
        return 0;
}
void odd_even_digit_count(int num,int *oc,int *ec)
{
        int rem;
        while(num!=0)
        {
                rem=num%10;
                if(rem%2==0)
```

```
                *ec=*ec+1;
            else
                *oc=*oc+1;
            num=num/10;
        }
    }
```

Output:

```
Enter an integer number:45287
Numbers of odd numbers=2
Number of even numbers=3
```

3. **Write a program to check whether a given word is palindrome or not using user defined function. [Hint: A palindrome word spells same back and forth e.g- madam]**

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
void check_palindrome(char str[]);
int main()
{
    char word[50];
    printf("Enter a word to check:");
    gets(word);
    check_palindrome(word);
    getch();
    return 0;
}
void check_palindrome(char str[])
{
    char A[50];
    strcpy(A,str);
    strrev(A);
    if(strcmp(str,A)==0)
        printf("%s is Palindrome",str);
    else
        printf("%s is not palindrome",str);

}
```

4. **Write a program to read a string. Now pass this string to a function that finds the number of vowels, consonants, digits, white spaces and other characters in the string. Your function must return these numbers to the calling function at once. In the calling function, display those numbers.**

```
#include<stdio.h>
#include<conio.h>
#include<ctype.h>
void characterscount(char str[],int count[]);
int main()
{
    char str[100];
    int count[5]={0,0,0,0,0};
    printf("Enter a string:");
    gets(str);
    characterscount(str,count);
```

```
        printf("Number of Vowels=%d",count[0]);
        printf("\nNumber of consonants=%d",count[1]);
        printf("\nNumber of digits=%d",count[2]);
        printf("\nNumber of white space=%d",count[3]);
        printf("\nNumber of other characters=%d",count[4]);
        getch();
        return 0;
}
void characterscount(char str[],int count[])
{
        int i;
        for(i=0;str[i]!='\0';i++)
        {
                if(isalpha(str[i]))
                {

                        if(str[i]=='a'||str[i]=='e'||str[i]=='i'||str[i]=='o'||str[i]=='u'||str[i]=='A'||str[i]=='E'||str[i]=='I'||
                        str[i]=='O'||str[i]=='U')
                            count[0]++;
                        else
                            count[1]++;
                }
                else if(isdigit(str[i]))
                        count[2]++;
                else if(isspace(str[i]))
                        count[3]++;
                else
                        count[4]++;
        }
}
```

5. **Write a Program to input a square matrix and print the menu:**
   Menu
   1 Sum of ith now
   2 Sum of jth Column
   3 Sum of diagonal elements from left
   4 Sum of diagonal elements from right
   5 Sum of all elements
   6 Exit from program
   and perform tasks as per user's choice using user defined function repeatedly until the choice is exit (6).

```
        #include<stdio.h>
        #include<conio.h>
        int ithrowsum(int A[][10], int m);
        int jthcolumnsum(int A[][10], int m);
        void leftdiagonalsum(int A[][10],int m);
        void rightdiagonalsum(int A[][10], int m);
        void allsum(int A[][10], int m);
        int main()
        {
                int A[10][10],m,i,j,ch;
                printf("Enter number of row or column of square matrix:");
                scanf("%d",&m);
                for(i=0;i<m;i++)
                {
```

```
                for(j=0;j<m;j++)
                {
                        printf("Enter %d%dth element:",i,j);
                        scanf("%d",&A[i][j]);
                }
        }
        printf("Entered matrix");
        for(i=0;i<m;i++)
        {
                printf("\n");
                for(j=0;j<m;j++)
                {
                        printf("%d\t",A[i][j]);
                }
        }
        do{
                        printf("\nEnter Choice");
                        printf("\n1.      Sum of ith row");
                        printf("\n2.      Sum of jth Column");
                        printf("\n3.      Sum of diagonal elements from left");
                        printf("\n4.      Sum of diagonal elements from right");
                        printf("\n5.      Sum of all elements");
                        printf("\n6.      Exit from program \n");
                        scanf("%d",&ch);
                        switch(ch)
                        {
                                case 1:
                                        ithrowsum(A,m);
                                        break;
                                case 2:
                                        jthcolumnsum(A,m);
                                        break;
                                case 3:
                                        leftdiagonalsum(A,m);
                                        break;
                                case 4:
                                        rightdiagonalsum(A,m);
                                        break;
                                case 5:
                                        allsum(A,m);
                                        break;
                                case 6:
                                        break;
                                default:
                                        printf("Invalid Choice\n");

                        }
                }while(ch!=6);
        getch();
        return 0;
}
int ithrowsum(int A[][10], int m)
{
        int i,r,s=0;
```

```
	printf("Enter the row number to sum:");
	scanf("%d",&r);
	if(r>=m)
	{
		printf("There isnot row number %d",r);
		return 0;
	}
	for(i=0;i<m;i++)
	{
		s=s+A[r][i];
	}
	printf("sum of %dth row=%d",r,s);
	return 1;
}
int jthcolumnsum(int A[][10], int m)
{
	int i,r,s=0;
	printf("Enter the column number to sum:");
	scanf("%d",&r);
	if(r>=m)
	{
		printf("There isnot column number %d",r);
		return 0;
	}
	for(i=0;i<m;i++)
	{
		s=s+A[i][r];
	}
	printf("sum of %dth column=%d",r,s);
	return 1;
}
void leftdiagonalsum(int A[][10],int m)
{
	int i,j,s=0;

	for(i=0;i<m;i++)
	{
		for(j=0;j<m;j++)
		{
			if(i==j)
			{
				s=s+A[i][j];
			}
		}
	}
	printf("sum of diagonal form left=%d",s);
}
void rightdiagonalsum(int A[][10], int m)
{
	int i,j,s=0;
	for(i=0;i<m;i++)
	{
		for(j=0;j<m;j++)
		{
```

```
                        if(i+j==m-1)
                        {
                                s=s+A[i][j];
                        }
                }
        }

        printf("sum of diagonal form right=%d",s);
}
void allsum(int A[][10], int m)
{
        int i,j,s=0;

        for(i=0;i<m;i++)
        {
                for(j=0;j<m;j++)
                {
                        s=s+A[i][j];
                }
        }
        printf("sum of all element=%d",s);
}
```

7 Pointers

7.1 Introduction

- A pointer is a derived data type in 'C'. It is built from any one of the primary data type available in 'C' programming language. Pointers contain memory addresses as their values. Since these memory addresses are the locations in the computer memory where the program instructions and data are stored. Pointers can be used to access and manipulate data in memory.

Advantages of Pointers

- Pointers are most efficient in handling arrays.
- They can be used to return multiple values from function through function argument.
- They permit references to functions and thereby they provide facility to pass function as argument to another function.
- The use of pointer arrays to character string saves the data storage space in the memory.
- Pointers allow C to supports dynamic memory management.
- They provide an efficient tool for manipulating dynamic data structures such as structures, link lists, quos, stacks, trees and graphs.
- Pointers reduce the complexity and length of the program.
- Pointers increase the execution speed and thus reduce the program execution time. Of course the real power of C lies in the proper use of pointers.

Understanding Pointers

Computer memory is sequential collection of storage cells as shown in figure. Each cell is commonly known as byte and has a number called as address associated with it. Typically the addresses are numbered sequentially starting from zero. The last address depends upon the memory size. A computer having 64KB memory will have its last address as 65535.

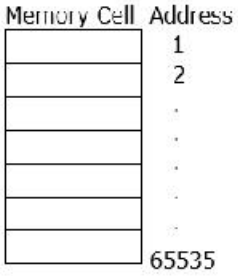
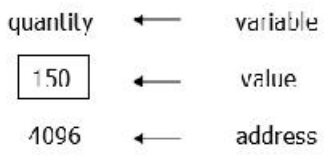


Figure 7-1: Memory Representation

Whenever we declare variable, the system allocates it somewhere in the memory. An appropriate location holds the value of variable. Since every byte has a unique address number, this location will have its own address number. Consider the following statement:

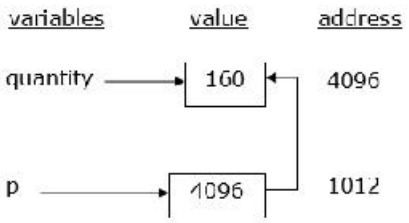
```
int quantity=150;
```

This statement instructs the system complier system to find a location for integer variable quantity and store value 150 in that location. Let us assume that the system has chosen the address location 4096 for quantity, so it will be visualized as:

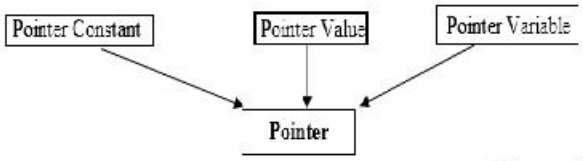


During the execution of program computer always associates the name quantity with address 4096. We may have access to the value 150 by using either the name quantity or the address 4096. Since memory addresses are simply numbers they can be assigned to some variables which can be stored in memory like any other variables. Such variables that hold memory addresses are called as 'pointer variables' the pointer variable is nothing but a variable that contains an address which is location of another variable in memory.

Remember: pointer is a variable. Its value is stored in memory in another location. Suppose we assign the address of quantity to a variable 'p', the link between the variable 'p' and quantity can be visualized as:



Since the value of variable 'p' is the address of variable quantity for using the value of 'p' and therefore we say that the variable p points to the variable quantity. Thus 'p' is called as pointer



The memory addresses within a computer are referred as pointer constants. We cannot change them; we can only use them to store the data values. The value that is stored as the memory location pointed by pointer is known as pointer value. It can be changed. The pointer value can be stored in another variable. The variable that contains the pointer value is called as pointer variable.

Accessing the address of variable

The actual location of variable in the memory is computer dependent and therefore the address of variable is unknown to immediately. If we want to identify address of any variable we can use the unary operator & of C programming language. It is known as 'address of' operator.

Example:

```
int x=10;
printf("%u",&x);
```

These statements will print the address of variable x where it is stored in the memory. The & operator can only be used with a simple variable or an array element.

7.2 Declaring pointer variable

- In C every variable must be declared for its data type. Since pointer variables contain addresses that belong to a separate data type they must be declared as pointers. The declaration of pointer variable takes following form:

```
type *ptrname;
```

Where type is a base type of the pointer and may be any valid data type and ptrname is a pointer variable and asterisk(*) is indirection operator .

- The statement tells that the variable-name is a pointer variable of data-type which will store address of the same data-type variable. For example:

```
int *p;
```

- It declares the variable 'p' as a pointer that will point to an integer data type variable. The p can store address of another integer variable only. Similarly,

```
float *x;
```

It declares x as pointer variable a floating point type.

7.3 Initialization of pointer variable

- The process of assigning the address of a variable to a pointer variable is known as initialization. For example:

```
int *p;
int x = 10;
p = &x;
```

7.4 Indirection or dereference Operator

- The operator '*', used in front of a variable, is called indirection or dereference operator. Normal variable provides direct access to their own values where as a pointer provides indirect access to the values of the variable whose address it stores.
- Example:

```
int *ptr,x=10;
p=&x;
printf("x=%d",*ptr);      //It gives x=10
*ptr=20 ;                 //indirectly it changes the value of x to 20
```

7.5 Chain of pointers (multiple indirection)

- A pointer can point to the address of another pointer.
- Example

```
int x=10,*p1,**p2;      //p2 is preceded by double indirection operator since it is capable of holding the address of integer pointer. [pointer-to-pointer]
p1=&x;
p2=&p1;                 //p2 holds the address of p1.
```

Example7-1:

```
#include<stdio.h>
#include<conio.h>
main()
{
        int x=10,*p1,**p2;
        p1=&x;
        p2=&p1;
        printf("%d",**p2);          //it gives output 10
        getch();
}
```

7.6 Pointer Arithmetic

- Pointer is a variable. Some arithmetic operations can be performed with pointers.
- We can add an integer to pointer, subtract an integer from pointer and subtract two pointer of same type. These operations are meaningless unless the pointer variable points to an array element. Besides these no other operations are allowed

- ***Pointer increment and decrement***

- Integer, float, char, double data type pointers can be increment and decremented. For all these data types both prefix and postfix increment or decrement is allowed.
- Integer pointers are incremented or decremented in the multiples of two. Similarly character by one, float by four and double pointers by eight etc
- E.g.

```
int *p;
p++          /*valid*/
++p          /*valid*/
p--          /*valid*/
--p          /*valid*/
```

Example 7-2:

```
#include<stdio.h>
#include<conio.h>
int main()
{
        int *p1,p;
        float *f1,f;
        char *c1,c;
        p1=&p;
        f1=&f;
        c1=&c;
        printf("Memory address before increment:\n");
        printf("int=%u\nfloat=%u\nchar=%u\n",p1,f1,c1);
        p1++;
        f1++;
        c1++;
        printf("Memory address after increment:\n");
        printf("int=%u\nfloat=%u\nchar=%u\n",p1,f1,c1);
        getch();
        return 0;
}
```

- ***Pointers addition and subtraction***
  - Other than addition and subtraction, no other operations are allowed on integer pointers. Addition and substraction with float or double data type pointers are not allowed.
  - Pointers cannot be added to each other. For example
    ```
    int *p1,*p2;
    p1=p1+p2;          /*invalid*/
    ```
  - We can subtracted one pointer from another in order to find the number of objects of their base type that separate the two. The two pointers must be of same type.

    E.g.-

    ```
    #include<stdio.h>
    #include<conio.h>
    int main()
    {
        int array[]={1,2,3,4,5,6,7,8,9,10},*ptr1,*ptr2;
        ptr1=&array[0];
        ptr2=&array[9];
        printf("ptr1=%u\nptr2=%u",ptr1,ptr2);
        printf("\nptr2-ptr1=%d",ptr2-ptr1);
        getch();
        return 0;
    }
    ```
  - A constant can be added or subtracted from integer pointer variable. For example
    ```
    int *ptr;
    ptr=ptr+9;
    ```

Example 7-3:

```
#include<stdio.h>
#include<conio.h>
int main()
{
        int x;
        int *ptr1,*ptr2;
```

```
        x=10;
        ptr1=&x;
        ptr2=ptr1+6;
        printf("Value of x=%d\n",x);
        printf("value pointed by ptr1=%d\n",*ptr1);
        printf("Address of x pointed by ptr1=%u\n",ptr1);
        printf("Content of ptr2=(ptr1+6) is=%u\n",ptr2);
        printf("Value pointed by ptr2=%u",*ptr2);
        getch();
        return 0;
}
```

- ***Pointer comparison***
  - Two pointers variables can be compared provided both pointers point to the element of the same data type.
- ***Pointer Multiplication and Division***
  - Multiplication or Division is not allowed with the pointers. For example

```
int *p1,*p2;
p1=p1*p2;        /*invalid*/
p1=p1/p2;        /*invalid*/
```

  - Pointers variable cannot be multiplied or divided by a constant. For example

```
int *p;
p = p*4;         /*invalid*/
p = p/2;         /*invalid*/
```

7.7 Rules for pointer operation

1. A pointer variable can be assigned address of another variable.
2. Pointer variable can be assigned value of another pointer variable.
3. Pointer variable can be initialized by NULL value of zero value.
4. Pointer variable can be used with pre increment or decrement or post increment or decrement operators.
5. An integer value may be subtracted or added from the pointer variable.
6. When two pointers point to the same array one pointer variable can be subtracted from another.
7. When two pointers point to the object of same data type we can use the relational operators to compare them.
8. Pointer variable cannot be multiplied or divided by a constant.
9. Two pointer variables cannot be added, subtracted, multiplied or divided.
10. A value cannot be assigned to any particular address. Example:

```
int a=20;
&a=1000;          // It is illegal.
```

7.8 Passing pointers to functions

A pointer can be passed to a function as an argument. Passing a pointer means passing address of a variable instead of value of the variable. As address is passed in this case, this mechanism is also k\a call by address or call by reference. When pointer is passed to a function, while function calling, the formal argument of the function must be compatible with the passing pointer i.e. if integer pointer is being passed, the formal argument in function must be pointer of the type integer and so on. As address of variable is passed in this mechanism, if value in the passed address is changed within function, the value of actual variable also changed.

Example7-4: Program to illustrate the use of passing pointer to a function

```
#include<stdio.h>
#include<conio.h>
void addGraceMarks(int *m)
{
        *m = *m+10;
}
```

```
int main()
{
        int marks;
        printf("Enter actual marks:");
        scanf("%d",&marks);
        addGraceMarks(&marks);                    //Passing address
        printf("\nTotal marks is: %d", marks);
        getch();
        return 0;
}
```

Example7-5: Program to convert upper case letter into lower and vice versa using passing pointer to a function

```
#include<stdio.h>
#include<conio.h>
void conversion(char*);
int main()
{
        char input;
        printf("Enter character of our choice: ");
        scanf("%c",&input);
        conversion(&input);
        printf("\n The corresponding character is: %c",input);
        getch();
        return 0;
}
void conversion(char *c)
{
        if(*c>=97 && *c<=122)
                *c=*c-32;
        else if(*c>=65 && *c<=90)
                *c=*c+32;
}
```

7.9 Function returning pointers

- Since pointers are a data type in C, we can also force a function to return a pointer to the calling function.

Example 7-6:

```
#include<stdio.h>
#include<conio.h>
int *larger(int*, int*);
main()
{
        int a=10;
        int b=20;
        int *ptr;
        ptr=larger(&a,&b);
        printf("%d",*ptr);
}
int *larger(int *x, int *y)
{
        if(*x>*y)
                return (x);                    //address of a
        else
                return (y);                    //address of b
}
```

- The function **larger** receives the address of the variable **a** and **b**, decides which one is larger using the pointers **x** and **y** and then return the address of its location. The return value is then assigned to the pointer variable **ptr** in the calling function.
- Note that the address returned must be the address of a variable in the calling function. It is an error to return a pointer to a local variable in the called function.

7.10 Pointers and arrays

- When an array is declared the compiler allocates base address and sufficient amount of storage memory to contain all the elements of the array in contiguous memory location. The base address is the location first element that is 0th element of the array. The compiler also defines the array name as a constant pointer to the first element.
- For example:

```
int x[5]={8,4,9,6,3};
```

Suppose the base address of x is 9092 and assuming each integer requires 2 bytes the five elements are stored as shown below

Index	Value	Address
x[0]	8	9092
x[1]	4	9094
x[2]	9	9096
x[3]	6	9098
x[4]	3	9100

The name x is defined as a constant pointer pointing to the first element x[0] and therefore value of x is 9092. If we declare p as an integer pointer then we can make the pointer p to point the array x by following assignment.

```
int *p;
p = &x[0];
OR
p = x;
```

Now we can access every value of x using p++ to move from first element to another

Thus ***(array + index)** is equivalent to **array[index].**

Example7-7: Program to find the average age of five students using pointers

```
#include<stdio.h>
#include<conio.h>
int main()
{
        int x[5],*p,sum=0,i;
        float avg;
        for(i=0;i<5;i++)
        {
                printf("Enter age of %d Student:",i+1);
                scanf("%d",&x[i]);
        }
        p=x;
        for(i=0;i<5;i++)
        {
         sum=sum+*(p+i);
        }
        avg=sum/5.0;
        printf("Average=%f",avg);
        getch();
        return 0;
}
```

7.11 Pointers to String

Strings are created like character array. Therefore, they are declared and initialized as:

char city[10] = "KATHMANDU";

K	A	T	H	M	A	N	D	U	\0

Compiler automatically inserts NULL character '\0' at the end of a string. 'C' supports an alternative method to create strings using pointer variables of type 'char'. Such as,

char *city= "KATHMANDU";

This creates a string and then stores its address in the pointer variable city. This pointer variable now points to the first character of the string. That is, the statements:

printf("%c",*city);

will print the current pointing character 'K' on the screen and the statement,

printf("%s", city);

will print all the sets of characters from current pointing location to the first occurrence of '\0'.

7.12 Pointer and Multidimensional Array:

We know that the name of array is point to first element of array i.e. 0th element. In two dimensional array the array point to [0] [0] element. In three dimensional array the array name point to [0] [0] [0] element, etc.

In two dimensional array the second element is [0] [1].

If we want to evaluate the [1] [2] element through pointer then the expression for that is *(*(p+1)+2); where p is pointer to array (i.e. p contains the address of [0] [0] element) or p is name of array.

Example7-8:

```
#include<stdio.h>
#include<conio.h>
main()
{
        int a[2][3]={{5,6,7},{8,9,10}};
        int i,j;
        for(i=0; i<2;i++)
        {
                for(j=0;j<3;j++)
                {
                        printf("%d\t",*(*(a+i)+j));
                }
                printf("\n");
        }
        getch();
}
```

7.13 Array of Pointers

- The way there can be an array of **int**s or an array of **float**s, similarly, there can be an array of pointers. Since a pointer variable always contain an address, an array of pointers would be nothing but a collection of addresses.
- The addresses present in the array of pointers can be address of variable or addresses of array element.
- We can create the array of pointers also with the same declaration of the array. For example:

```
int *A[15];
```

here, the array of pointers will be created i.e. the array will point to 15 different integers, such as:

```
A[0]=&x;
A[1]=&y;
```

```
A[2]=&z;
.
.
.
```

Example7-9:

```
#include<stdio.h>
#include<conio.h>
int main()
{
    int *A[4],i,x,y,z,sum=0;
    A[0]=&x;
    A[1]=&y;
    A[2]=&z;
    A[3]=&sum;
    for(i=0;i<3;i++)
    {
        printf("Enter %d number:",i+1);
        scanf("%d",A[i]);
    }
    for(i=0;i<3;i++)
    {
        *A[3]=*A[3]+*A[i];
    }
    printf("Sum=%d",*A[3]);
    getch();
    return 0;
}
```

- **char city[3][15];**

This declaration says that '**city**' is a table containing 3 names each with maximum length of 15 characters. So, the total storage requirement for this array is 45 bytes. We know that very rarely the individual string will be of equal lengths. Therefore instead of making each row of a fixed number of characters we can make it a pointer to a string of variable length. Example:

char *city[3]={"Kathmandu", "Pokhara", "Biratnagar"};

This declares **city** to be an array of three pointers each pointing to particular city names as:

city[0] "Kathmandu"
city[1] "Pokhara"
city[2] "Biratnagar"

This declares declaration allocates only 29 bytes of memory to store the string. Following statement will print the strings on the screen:

```
for(i=0;i<3;i++)
  printf("\n %s", city[i]);
```

7.14 Dynamic Memory Allocation (DMA):

- The process of allocating and freeing memory at run time is known as Dynamic Memory Allocation. This reserves the memory required by the program and returns this resource to the system once the use of reserved space utilized.
- Through arrays can be used for data storage, they are of fixed size. The programmer must know the size of the array or data in advanced while writing the program. In some cases, it is not possible to know the size of the memory required well ahead and to keep a lot of memory reserved is not also good practice. In such situation, DMA will be useful.
- Since an array name is actually a pointer to the first element within the array, it is possible to define the array as a pointer variable to represent an array requires some type of initial memory assignment before the array elements are processed. This is known as DMA. DMA refers allocating and freeing memory at run time.
- There are 4 library functions malloc( ), calloc( ), free( ) and realloc( ) for memory management. These functions are defined within header file **stdlib.h** and **alloc.h**

1. **malloc( )**

It allocates requested size of bytes and returns a pointer to the first byte of the allocated space. Its syntax is as

ptr=(data_type*)malloc(size_of_block);

where ptr is a pointer of type data_type. The malloc( ) returns a pointer to an area of memory with size size_of_block. For example:

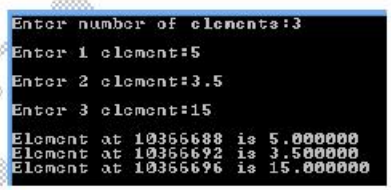
x=(int*)malloc(100*sizeof(int));

A memory space equivalent to 100 times the size of an integer (i.e. 100*2 bytes = 200 bytes) is reserved and the address of the first byte of the memory allocated is assigned to the pointer x of type int (i.e. x refers the first address of allocated memory).

Example 7-10: This program reads array elements and display them with their corresponding storage locations.

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
main()
{
        int n,i;
        float *address;
        printf("Enter number of elements:");
        scanf("%d",&n);
        address=(float*)malloc(n*sizeof(float));
        if(address==NULL)
        {
                printf("No space available\n");
                exit(1);
        }
        for(i=0;i<n;i++)
        {
                printf("\nEnter %d element:",i+1);
                scanf("%f",(address+i));
        }
        for(i=0;i<n;i++)
        {
                printf("\nElement at %u is %f",(address+i),*(address+i));
        }
        free(address);
        getch();
}
```

Output:



Note: *Sometimes memory cannot be allocated due to lack of requested amount of free memory. We must handle this situation so, we should check whether the required memory is allocated or not in every program. If the memory cannot be allocated, the memory allocation function returns NULL pointer to the pointer variable.*

2. **calloc( )**

The function calloc( ) provides access to the C memory heap which is available for dynamic allocation of variable_size block of memory. Unlike malloc( ), the function calloc( ) accepts two arguments: no_of_blocks and size_of_block. This parameter no_of_blocks specifies the number of items to allocate and size_of_block specifies the size of each item. The function calloc( ) allocates multiple blocks of storage, each of the same size and then sets all bytes to zero.

One important difference between malloc( ) and calloc( ) is that calloc( ) initializes all bytes in the allocated block to zero. Thus, it is normally used for requesting memory space at runtime for storing derived data type such as arrays user defined. Its syntax is

ptr=(data_type*)calloc(no_of_block,size_of_each_block);

For example:

x=(int*)calloc(5,10*sizeof(int));

or

x=(int*)calloc(5,20);

The above statement allocates contiguous space for 5 blocks, each of size 20 bytes i.e. we can store 5 arrays, each of 10 elements of integer types.

3. **realloc()**

This function is used to modify the size of previously allocated space. Sometimes, the previously allocated memory is not sufficient, we need additional space and sometime the allocated memory is much larger than necessary. In both situations, we can change the memory size already allocated with the help of function **realloc( )**. Its syntax is as

If the original allocation is done by the statement,

ptr=malloc(size);

then reallocationof space may be done by the statement

ptr=realloc(ptr,newsize);

This function allocates a new memory space of new size to the pointer variable ptr and returns a pointer to the first byte of new memory block and on failure the function return NULL.

4. **free( )**

The built-in function frees previously allocated space by calloc, malloc or realloc function. The memory dynamically allocated is not returned to the system until the programmer returns the memory explicitly. This can be done using **free( )** function. Thus, this function is used to release the space when it is not required. Its syntax is

free(ptr);

Where ptr is a pointer to a memory block which has already been created by **malloc( )**, **calloc( )** or **realloc( )**function.

❖ **Memory leak:**

- It is often thought of as a failure to release unused memory by a computer program.
- It is just unnecessary memory consumption.
- A memory leak diminishes the performance of the computer, as it becomes unable to use all its available memory.
- To avoid memory leaks, memory allocated on heap should always be freed when no longer needed.

7.15 Two-dimensional dynamic memory allocation

- Two-dimensional dynamic memory allocation is similar to one dimensional dynamic memory allocation. We should know that a two dimensional array is an array of one dimensional arrays. For two dimensional memory allocation, it is required to allocate memory for an array of pointers. Each elements of the array holds the starting address of each row of the two dimensional array.
- If we want to allocate memory for a two dimensional array of size **rows** X **cols**, we have to declares a pointer variable that points to pointer.
- For example, let us take an example of to allocate memory for a 4 X 4 matrix. Let us declare an identifier matrix as a pointer variable that points to an array of pointers. Each member of the array points to each array of 4 X 4 array in one to one fashion.

float **matrix;

Where matrix is a pointer to point to an array of float pointers. To create an array of pointers of size 4, **calloc** function can be used as in the following syntax.

matrix=(float **)calloc(4,sizeof(float);

Here **calloc** function allocates memory for an array of size 4 and returns the base address (starting address) to matrix. To allocate memory for each row of the array, **calloc** function can be used again as

```
for(i=0;i<4;i++)
{
        matrix[i]=(float*)calloc(4,sizeof(float));
}
```

Here, in the first round of loop, **calloc** function allocates memory for the first row and returns the base address to matrix[0]. Similarly for second, third and fourth rows.

Address of element at i^{th} row and j^{th} column is (*(matrix+i)+j) and value at the location pointed by the address is *(*(matrix+i)+j).

Example 7-11: A program to use function to read, process and display the sum of two dimensional dynamic arrays

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
void readmatrix(float**,int,int);
void displaymatrix(float**,int,int);
void processmatrix(float**,float**,int,int);
main()
{
        int r1,c1,r2,c2,i;
        float **matrix1,**matrix2;
        printf("Enter rows of matrix1:");
        scanf("%d",&r1);
        printf("Enter columns of matrix1:");
        scanf("%d",&c1);
        printf("Enter rows of matrix2:");
        scanf("%d",&r2);
        printf("Enter columns of matrix2:");
        scanf("%d",&c2);
        /*allocating memory for matrix1*/
        matrix1=(float**)calloc(r1,sizeof(float));
        for(i=0;i<r1;i++)
        {
                matrix1[i]=(float*)calloc(c1,sizeof(float));
        }
        /*allocating memory for matrix2*/
        matrix2=(float**)calloc(r2,sizeof(float));
        for(i=0;i<r2;i++)
        {
                matrix2[i]=(float*)calloc(c2,sizeof(float));
        }
        printf("Read Matrix1:\n");
        readmatrix(matrix1,r1,c1);
        printf("Read Matrix2:\n");
        readmatrix(matrix2,r2,c2);
        printf("Matrix1:\n");
        displaymatrix(matrix1,r1,c1);
        printf("Matrix2:\n");
        displaymatrix(matrix2,r2,c2);
        if(r1==r2&&c1==c2)
        {
                processmatrix(matrix1,matrix2,r1,c1);
                printf("Resultant Matrix:\n");
                displaymatrix(matrix1,r1,c1);
```

```
        }
        else
        {
                printf("The order of matrix are not same!!");
        }
        free(matrix1);
        free(matrix2);
        getch();
}
void readmatrix(float** mat,int r,int c)
{
        int i,j;
        printf("Enter matrix elements\n");
        for(i=0;i<r;i++)
        {
                for(j=0;j<c;j++)
                {
                        scanf("%f",(*(mat+i)+j));
                }
        }
}
void displaymatrix(float** mat,int r,int c)
{
        int i,j;
        for(i=0;i<r;i++)
        {
                for(j=0;j<c;j++)
                {
                        printf("%.2f\t",*(*(mat+i)+j));
                }
                printf("\n");
        }
}
void processmatrix(float** mat1,float** mat2,int r,int c)
{
        int i,j;
        for(i=0;i<r;i++)
        {
                for(j=0;j<c;j++)
                {
                        *(*(mat1+i)+j)=*(*(mat1+i)+j)+*(*(mat2+i)+j);
                }
        }
}
```

Output:

```
Enter rows of matrix1:2
Enter columns of matrix1:2
Enter rows of matrix2:2
Enter columns of matrix2:2
Read Matrix1:
Enter matrix elements:
1       2
3       4.5
Read Matrix2:
Enter matrix elements:
3       4
2.5     5
Matrix1:
1.00    2.00
3.00    4.50
Matrix2:
3.00    4.00
2.50    5.00
Resultant Matrix:
4.00    6.00
5.50    9.50
```

Void Pointer

- Suppose we have to declare integer pointer, character pointer and float pointer then we need to declare 3 pointer variables.
- Instead of declaring different types of pointer variable it is feasible to declare single pointer variable which can act as integer pointer, character pointer and float pointer.
- But in exchange they have a great limitation: the data pointed by them cannot be directly dereferenced, and for that reason we will always have to change the type of the void pointer to some other pointer type that points to a concrete data type before dereferencing it. This is done by performing type-casting.
- We cannot apply pointer arithmetic to void pointers.

Declaration of Void Pointer:

```
void * pointer_name;
```

Example:

```
#include<stdio.h>
#include<conio.h>
int main()
{
    void *ptr;    // ptr is declared as Void pointer
    char ch='g';
    int num=5;
    float fnum=10.5;
    ptr = &ch;  // ptr has address of character data
    printf("Charactered is: %c",*(char*)ptr);
                        /* Converting void pointers to character pointer using type casting(char*) */
    ptr = &num;  // ptr has address of integer data
    printf("\nInteger number is: %d",*(int*)ptr);
                        /* Converting void pointers to integer pointer using type casting(int*) */
    ptr = &fnum;  // ptr has address of float data
    printf("\nFloating point number is: %f",*(float*)ptr);
                        /* Converting void pointers to float pointer using type casting(float*) */
    getch();
    return 0;
}
```

Output:

```
Charactered is: g
Integer number is: 5
Floating point number is: 10.500000
```

NULL Pointer

- NULL Pointer is a pointer which is pointing to nothing.
- NULL pointer points the base address of segment.
- In case, if you don't have address to be assigned to pointer then you can simply use NULL
- Pointer which is initialized with NULL value is considered as NULL pointer.
- NULL is macro constant defined in following header files –

  stdio.h, alloc.h, mem.h, stddef.h, stdlib.h

Example:

```
#include <stdio.h>
#include<conio.h>
int main()
{
        int  *ptr = NULL;
        printf("The value of ptr is %u",ptr);
        getch();
        return 0;
}
```

Output:

The value of ptr is 0

Bad Pointer

- When a pointer is first allocated, it does not have a pointee(valid address). The pointer is "uninitialized" or simply "bad". A dereference operation on a bad pointer is a serious runtime error.
- It's always a best practice to initialize a Pointer to NULL value when they are declared and check for whether the pointer is a NULL pointer when using the pointer.
  i.e., if you declare a pointer and don't yet know where to point the pointer, then initialize the pointer to NULL.

Some Solved Example

1. **Write a C program using pointer to reverse an array.**

```
#include <stdio.h>
#include <conio.h>
#include<stdlib.h>
int main()
{
    int *ptr,i,j,n,temp;
    printf("Enter the number of elements in the array: ");
    scanf("%d",&n);
    ptr=(int *)malloc(n*sizeof(int));
    if(ptr==NULL)
    {
        printf("Memory cannot be created");
        exit(1);
     }
    printf("Enter element:\n");
    for(i=0;i<n;i++)
        scanf("%d",ptr+i);
    for(i=0,j=n-1;i<n/2;i++,j--)
    {
        temp = *(ptr+i);
        *(ptr+i) = *(ptr+j);
        *(ptr+j) = temp;

    }
    printf("Reversed array elements are:\n");
    for(i=0;i<n;i++)
    printf("%d\t",*(ptr+i));
    free(ptr);
    getch();
    return 0;
}
```

Output:

```
Enter the number of elements in the array: 10
Enter element:
7       5       6       4       2       3       8       1       9       2
Reversed array elements are:
2       9       1       8       3       2       4       6       5       7
```

2. **Write a program using pointer to arrange characters in a word in alphabetical order**

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
int main()
{
    char word[100],temp,*str;
    int i,j;
    printf("Enter a word:");
    gets(word);
    str=word;
    for(i=0;*(str+i)!='\0';i++)
    {
```

```
                for(j=i+1;*(str+j)!='\0';j++)
                {
                        if(*(str+i)>*(str+j))
                        {
                                temp=*(str+i);
                                *(str+i)=*(str+j);
                                *(str+j)=temp;
                        }
                }
        }
        printf("Alphabetical order:\n");
        puts(str);
        getch();
        return 0;

}
```

3. Write a program with user defined function using pointer to convert all the upper case to lower case and vice-versa in a string given by the user

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
#include<string.h>
#include<ctype.h>
void conversion(char *ptr);
int main()
{
        char str[100];
        printf("Enter string:");
        gets(str);
        conversion(str);
        puts(str);
        getch();
        return 0;
}
void conversion(char *ptr)
{
        int i;
        for(i=0;*(ptr+i)!='\0';i++)
        {
                if(islower(*(ptr+i)))
                        *(ptr+i)=toupper(*(ptr+i));
                else
                        *(ptr+i)=tolower(*(ptr+i));
        }
}
```

4. This Program sorts a dynamic array using concept of both function and pointer.

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
void sort(float [],int);
```

```
int main()
{
	float *marks;
	int n,i;
	printf("Enter the number of elements in the array: ");
	scanf("%d",&n);
	marks=(float *)malloc(n*sizeof(float));
	if(marks==NULL)
	{
		printf("Memory cannot be created");
		exit(1);
	}
	printf("Enter element:\n");
	for(i=0;i<n;i++)
		scanf("%f",marks+i);
	sort(marks,n);
	printf("Sorted array:\n");
	for(i=0;i<n;i++)
		printf("%f\t",*(marks+i));
	free(marks);
	getch();
	return 0;
}
void sort(float *ptr,int n)
{
	int i,j;
	float temp;
	for(i=0;i<n-1;i++)
	{
		for(j=0;j<n-i-1;j++)
		{
			if(*(ptr+j)>*(ptr+j+1))
			{
				temp=*(ptr+j+1);
				*(ptr+j+1)=*(ptr+j);
				*(ptr+j)=temp;
			}
		}
	}
}
```

5. **Write a program to read an array of n elements. The program has to allocate memory dynamically and pass the array to a function, which has to find the smallest, the largest numbers and average of all elements and pass to the calling function. Use pass by reference if necessary.**

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
void findminmax(int,int*,int*,int*,float*);
int main()
{
	int *array,n,largest,smallest,i;
	float average;
	printf("Enter number of element in the array:");
	scanf("%d",&n);
```

```
        array=(int*)calloc(n,sizeof(int));
        if(array==NULL)
        {
                printf("Memory cannot be created");
                exit(1);
        }
        for(i=0;i<n;i++)
        {
                printf("\nEnter array element[%d]:",i+1);
                scanf("%d",(array+i));
        }
        findminmax(n,array,&largest,&smallest,&average);
        printf("\nLargest=%d\nSmallest=%d\nAverage=%f",largest,smallest,average);
        free(array);
        getch();
        return 0;
}
void findminmax(int n,int *ptr,int *max,int *min,float *avg)
{
        int i,sum=*(ptr+0);
        *max=*(ptr+0);
        *min=*(ptr+0);
        for(i=1;i<n;i++)
        {
                sum=sum+*(ptr+i);
                if(*max<*(ptr+i))
                {
                        *max=*(ptr+i);
                }
                if(*min>*(ptr+i))
                {
                        *min=*(ptr+i);
                }
        }
        *avg=(float)sum/n;
}
```

Output:

```
Enter number of element in the array:5

Enter array element[1]:5

Enter array element[2]:4

Enter array element[3]:2

Enter array element[4]:8

Enter array element[5]:3

Largest=8
Smallest=2
Average=4.400000
```

8 Structures and Union

We know an array is used to store a collection of data of the same type. But if we want to deal with a collection of data of various type such as integer, string, float etc an array cannot be used. The solution for this problem is structures. It is a derived data-type in C, which is a collection of different datatype elements. It is a convenient tool for handling a group of logically related data items. Structure helps to organize the complex data in more meaningful way.

8.1 Defining a Structures

- Structures must be defined first for their format that may be used later to declare structure variables. The general format or syntax to create a structure is:

```
struct  tag_name
{
        data_type variable1;
        data_type variable1;
        ……….
        ……….
};
```

- Remember, this template is terminated by a semicolon. The keyword struct declares a structure to hold the details of different data elements. For example:

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

- Here, the fields: title, author, pages and price are called as **structure elements or members**. Each of these members belongs to different data types. The name of structure 'book' is called structure tag.

8.2 Declaring structure variables

- After declaring a structure format we can declare variables of that type. A structure variable declaration is similar to the declaration of variables of another type. It includes following elements:
    1. The keyword struct
    2. The structure tag name
    3. List of variable names separated by commas
    4. A terminating semicolon For example, the statement:

```
struct book book1, book2, book3;
```

- Declares book1, book2, book3 as variables of type struct book. Each of these variables has four members as specified by template. The declaration might look like this:

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

 struct book book1, book2, book3;
```

- The members of structure themselves are not variables. They do not occupy any memory until they are associated with structure variables such as book1. It is also allowed to combine both the structure definition and variables declaration in one statement.

The declaration,

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

}book book1, book2, book3;
```

Is valid. Use of tag name is also optional. For example:

```
struct {
        char title[20];
        char author[15];
        int pages;
        float price;
}book1, book2, book3;
```

- It declares book1, book2, book3 as structure variables representing three books but does not include tag name.

8.3 Accessing Structure Variables

- There are two types of operators to access members of a structure which are:
  - Member operator (dot operator or period operator(.))
  - Structure pointer operator(->)

8.3.1 Member operator (.)

- The link between member and variable is established using member operator when structure members are accessed by using structure variables. Any member of a structure can be accessed as:

  Structure_variable_name . member_name

- For example:

  book1.price

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

```
strcpy(book1.title, "Programming");
strcpy(book1.author, "T.B.Kute");
book1.pages = 375;
book1.price = 275.00;
```

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

```
scanf("%s", book1.title);
scanf("%s", book1.author);
scanf("%d",&book1.pages);
scanf("%f", &book1.price);
```

are valid input statements.

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

```
printf("%s", book1.title);
printf("%s", book1.author);
printf("%d",book1.pages);
printf("%f", book1.price);
```

8.3.2 Structure pointer operator (->)

- C provides the structure pointer operator -> to access members of a structure via a pointer. If a pointer variable is assigned the address of a structure, then a member of the structure can be accessed by:

  pointer_to_structure->member_name

8.4 Structure Initialization

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

```
struct time
{
        int hrs;
        int mins;
        int secs;
};
struct time t1 = {4, 52, 29};
struct time t2 = {10, 40, 21};
```

- This assigns value 4 to t1.hrs, 52 to t1.mins, 29 to t1.secs and value 10 to t2.hrs, 40 to t2.mins, 21 to t2.secs. There is one-to-one correspondence between the members and their initializing values.
- C does not permit the initialization of individual structure members within the template. The initialization must be done only in the declaration of actual variables.

Example 8-1: Program to declare the structure student having member variables roll_no, name and age. Accept and display data

```
#include<stdio.h>
#include<conio.h>
struct student
        {
                int roll_no;
                char name[10];
                int age;
        };
main()
{
        struct student s;
        printf("Enter Roll:");
        scanf("%d",&s.roll_no);
        printf("Enter Name:");
        scanf("%s",&s.name);
        printf("Enter Age:");
        scanf("%d",&s.age);
        printf("\nEntered information: \n");
        printf("Roll number: %d",s.roll_no);
        printf("\nName: %s",s.name);
        printf("\nAge: %d",s.age);
        getch();
}
```

Output:

```
Enter Roll:301
Enter Name:ram
Enter Age:20

Entered information:
Roll number: 301
Name: ram
Age: 20
```

8.5 Copying and comparing structure variables

Two variables of the same structure type can be copied the same way as ordinary variables. If **student1** and **student2** belongs to the same structure, then the following statements are valid:

```
student1 = student2;
student2 = student1;
```

However, the statements such as,

student1 == student2

student2 != student1

are not permitted. C does not permit any logical operation on structure variables. In case, we need to compare them, we may do so by comparing members individually.

Example 8-2: Write a program to illustrate the comparison and copying of structure variables

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
struct classes
{
        int roll;
        char name[20];
        float marks;
};
main()
{
        int x;
        struct classes student1={111,"Ram",75.5};
        struct classes student2={121,"Hari",65.0};
        struct classes student3;
        student3=student2;
        x=((student3.roll==student2.roll)&&(strcmp(student3.name,student2.name)==0)&&
        (student3.marks==student2.marks))?1:0;
        if(x==1)
        {
                printf("\nstudent2 and student3 are same\n");
                printf("%d\t%s\t%f",student3.roll,student3.name,student3.marks);
        }
        else
                printf("\nstudent2 and student3 are different\n");

        getch();
}
```

Output:

```
student2 and student3 are same
121     Hari    65.000000
```

8.6 Arrays of structures

- Like any other data type variable, array of structure variables can be declared. The array will have individual structures as its elements.

Example 8-3: This example declares an array of structure variable s having 3 members. Each member in turn is a structure having four members

```
#include<stdio.h>
#include<conio.h>
struct student
{
        char name[50];
        int age;
        int roll;
        char sec;
};
```

```
main()
{
        struct student s[3];    /*Declaring an array of structures*/
        int i;
        for(i=0;i<3;i++)
        {
                printf("Enter name of student:");
                scanf("%s",s[i].name);
                printf("Enter age:");
                scanf("%d",&s[i].age);
                printf("Enter roll:");
                scanf("%d",&s[i].roll);
                printf("Enter section:");
                fflush(stdin);
                scanf("%c",&s[i].sec);
        }
        printf("\nEntered Information:\n");
        for(i=0;i<3;i++)
        {
                printf("Name:%s\nAge:%d\nRoll:%d\nSec:%c\n",s[i].name,s[i].age,s[i].roll,s[i].sec);
        }
        getch();
}
```

8.7 Arrays within structures

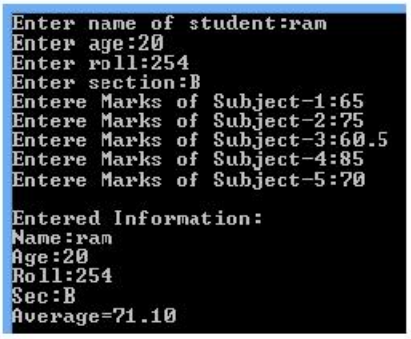
- C permits the use of arrays as structures members. We have already used array of characters. Similarly, we can use single or multidimensional array of other data type like int, float etc.

Example 8-4:

```
#include<stdio.h>
#include<conio.h>
struct student
{
        char name[50];
        int age;
        int roll;
        char sec;
        float marks[5];
};
main()
{
        struct student s;
        int i;
        float average,sum=0;
        printf("Enter name of student:");
        scanf("%s",s.name);
        printf("Enter age:");
        scanf("%d",&s.age);
        printf("Enter roll:");
        scanf("%d",&s.roll);
        printf("Enter section:");
        fflush(stdin);
        scanf("%c",&s.sec);
```

```
        for(i=0;i<5;i++)
        {
                printf("Entere Marks of Subject-%d:",i+1);
                scanf("%f",&s.marks[i]);
                sum=sum+s.marks[i];
}
        average=sum/5;
        printf("\nEntered Information:\n");
        printf("Name:%s\nAge:%d\nRoll:%d\nSec:%c\nAverage=%.2f",s.name,s.age,s.roll,s.sec,average);

        getch();
    }
```

Output:



8.8 Structures within structures

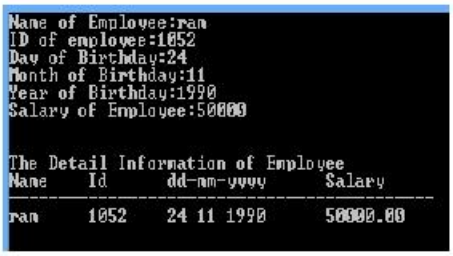
- Structures within structures means **nesting** of structures. The individual members of a structure can be other structure as well.

Example 8-5: Create a structure named date that has day, month and year as its members. Include this structure as a member in another structure named employee which has name, id, salary, as other members. Use this structure to read and display employee's name, id, dob and salary.

```
#include<stdio.h>
#include<conio.h>
struct date
{
        int day;
        int month;
        int year;
};
struct employee
{
        char name[20];
        int id;
        struct date dob;
        float salary;
}emp;
main()
{
        printf("Name of Employee:");
        scanf("%s",emp.name);
        printf("ID of employee:");
        scanf("%d",&emp.id);
```

```
        printf("Day of Birthday:");
        scanf("%d",&emp.dob.day);
        printf("Month of Birthday:");
        scanf("%d",&emp.dob.month);
        printf("Year of Birthday:");
        scanf("%d",&emp.dob.year);
        printf("Salary of Employee:");
        scanf("%f",&emp.salary);
        printf("\n\nThe Detail Information of Employee\n");
        printf("Name\tId\tdd-mm-yyyy\tSalary");
        printf("\n-----------------------------------------\n");
       printf("%s\t%d\t%d%d%d\t%.2f",emp.name,emp.id,emp.dob.day,emp.dob.month,emp.dob.year,emp.salary);
        getch();
}
```

Output:



8.9 Structures and Functions

- C supports passing of structures as arguments to functions. There are two methods by which the values of a structures can be transferred from one function to another.
  - Passing by value
  - Passing by reference

8.9.1 Passing by value

- This methods involve passing a copy of the entire structure to the called function. Any change to the structure members within the function are not reflected in the original structure (in the calling function). It is, therefore, necessary for the function to return the entire structure back to the calling function.

Example 8-6:

```
#include<stdio.h>
#include<conio.h>
void display(struct student);
struct student
{
        char name[50];
        int age;
        int roll;
        char sec;
};
main()
{
        struct student s;
        printf("Enter name of student:");
        scanf("%s",s.name);
        printf("Enter age:");
        scanf("%d",&s.age);
```

```
        printf("Enter roll:");
        scanf("%d",&s.roll);
        printf("Enter section:");
        fflush(stdin);
        scanf("%c",&s.sec);
        display(s);
        getch();
}
void display(struct student s)
{
        printf("\nEntered Information:\n");
        printf("Name:%s\nAge:%d\nRoll:%d\nSec:%c",s.name,s.age,s.roll,s.sec);
}
```

8.9.2 Passing by reference

- This method uses the concept of pointer to pass structure as an argument. The address location of the structure is passed to the called functions.
- The function can access indirectly the entire structure and work on it.
- Structure pointer operator is used to access the member.

Example 8-7:

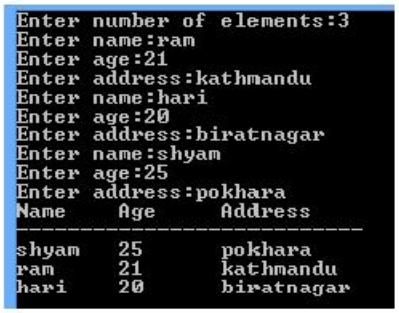
```
#include<stdio.h>
#include<conio.h>
void display(struct student*);
struct student
{
        char name[50];
        int age;
        int roll;
        char sec;
};
main()
{
        struct student s;
        printf("Enter name of student:");
        scanf("%s",s.name);
        printf("Enter age:");
        scanf("%d",&s.age);
        printf("Enter roll:");
        scanf("%d",&s.roll);
        printf("Enter section:");
        fflush(stdin);
        scanf("%c",&s.sec);
        display(&s);
        getch();
}
void display(struct student *ptr)
{
        printf("\nEntered Information:\n");
        printf("Name:%s\nAge:%d\nRoll:%d\nSec:%c",ptr->name,ptr->age,ptr->roll,ptr->sec);
}
```

Example 8-8: This example illustrates the concept of dynamic memory allocation for creating array of structure of size n In this example, a structure employee having members name, age and address is used. The array is passed to a function which sorts the array in descending order on the basis of member age and display the sorted array from the main.

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
struct employee
{
        char name[50];
        int age;
        char address[50];
};
void sort(struct employee*, int);
main()
{
        struct employee *p;
        int i,n;
        printf("Enter number of elements:");
        scanf("%d",&n);
        p=(struct employee*)calloc(n,sizeof(struct employee));
        if(p==NULL)
        exit(1);
        for(i=0;i<n;i++)
        {
                printf("Enter name:");
                scanf("%s",(p+i)->name);
                printf("Enter age:");
                scanf("%d",&(p+i)->age);
                printf("Enter address:");
                scanf("%s",(p+i)->address);
        }
        sort(p,n);
        printf("\nName\tAge\tAddress");
        printf("\n-------------------------\n");
        for(i=0;i<n;i++)
        {
                printf("%s\t%d\t%s\n",(p+i)->name,(p+i)->age,(p+i)->address);

        }
        getch();
}
void sort(struct employee *ptr,int num)
{
        struct employee temp;
        int i,j;
        for(i=0;i<num-1;i++)
        {
                for(j=0;j<num-1-i;j++)
                {
                        if((ptr+j)->age<(ptr+j+1)->age)
                        {
                                temp=*(ptr+j);
                                *(ptr+j)=*(ptr+j+1);
                                *(ptr+j+1)=temp;
```

```
                    }
                }
            }
        }
```

Output:



8.10 Self-referential structures

- A self-referential structures is a structure that contain a pointer to a structure of the same type.
- Self-referential structures are very useful in applications that involve linked data structures, such as lists and trees.
- It is sometimes desirable to include within a structure one member that is a pointer to the parent structure type.

In general terms, this can be expressed as

```
struct tag
{
        member 1;
        member 2;
        .........
        struct tag *name;
};
```

Where ***name*** refers to the name of a pointer variable. Thus, the structure of type **tag** will contain a member that points to another structure of type tag. Such structures are known as *self-referential* structures.

Example 8-9:

```
struct list_element
{
        char item[40];
        struct list_element *next;
};
```

This is a structure of type list_element. The structure contains two members, an array of character called item and a pointer to a structure of the same type called next. Therefore this is a self-referential structure.

8.11 Unions

- Both structures and unions are used to group a number of different variables together. Union is another way of creating user defined data types.
- Syntactically both structures and unions are almost same. The main difference is in storage. In structures, each member has its own storage location but all member of a union use the same memory location.
- Due to sharing of a common memory location, all the members of a union cannot be accessed at any time. The main objective of using union is to save memory space.
- Like structures, a union can be declared using the keyword **union.**

Definition

Structures	Union

struct student
{
char name[40];
int roll;
float marks;
}
	union student
	{
	char name[40];
	int roll;
	float marks;
	}

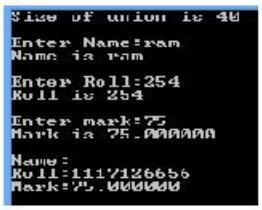
Memory allocation

Structures	40 bytes for name | 2 bytes for roll | 4bytes for marks
Union	40 bytes for name
Discussion	In case of structure memory is reserved for all the members separately but in case of union memory is reserved equal to the size of the largest member. Here, name is the largest member so only 40 bytes is reserved for the whole union.

Example 8-10:

```
#include<stdio.h>
#include<conio.h>
union student
{
        char name[40];
        int roll;
        float marks;
};
int main()
{
        union student s;
        printf("Size of union is %d",sizeof(s));
        printf("\n\nEnter Name:");
        scanf("%s",s.name);
        printf("Name is %s",s.name);
        printf("\n\nEnter Roll:");
        scanf("%d",&s.roll);
        printf("Roll is %d",s.roll);
        printf("\n\nEnter mark:");
        scanf("%f",&s.marks);
        printf("Mark is %f",s.marks);
        printf("\n\nName:%s\nRoll:%d\nMark:%f",s.name,s.roll,s.marks);
        /*In this case, only the value of marks is printed correctly all other members have garbage value*/
        getch();
        return 0;
}
```

Output:



Some Solved Examples

1. ***Write a program to maintain the records of different parts available in store using their*** *Part no, Part name, Stock quantity, Rate, Re-order level* ***using structure***

```
#include <stdio.h>
#include <conio.h>
struct parts
{
        int part_no;
        char part_name[50];
        int stock_quantity;
        int rate;
        int re_order;
};
main()
{
        struct parts s[100];
        int n,i;
        printf("Enter number of parts:");
        scanf("%d",&n);
        for(i=0;i<n;i++)
        {
                printf("Enter Detail of part %d:\n",i+1);
                printf("Enter part number:");
                scanf("%d",&s[i].part_no);
                printf("Enter name of part:");
                scanf("%s",&s[i].part_name);
                printf("Enter stock quantity:");
                scanf("%d", &s[i].stock_quantity);
                printf("Enter rate:");
                scanf("%d", &s[i].rate);
                printf("Enter Re-order Level:");
                scanf("%d", &s[i].re_order);
        }

        printf("\npart_no\tpart_name\tstock_quantity\trate\tre_order_level");
        for(i=0;i<n;i++)
        {
            printf("\n%d\t%s\t\t%d\t\t%d\t%d",s[i].part_no,s[i].part_name,s[i].stock_qantity,
                                                        s[i].rate,s[i].re_order);
        }
        getch();
}
```

2. ***Write a program that contains the customer's name (char), loan_number (int) and balance (float) of 100 customers in a bank and print the name that has the highest loan from the bank.***

```
#include<stdio.h>
#include<conio.h>
struct customer
{
        char name[50];
        int loan_number;
        float balance;
};
```

```
main()
{
        struct customer c[100];
        int n,i;
        float max=0;
        printf("Enter the number of customer");
        scanf("%d",&n);
        for(i=0;i<n;i++)
        {
                printf("Enter %dcustomer name",i+1);
                scanf("%s",&c[i].name);
                printf("Enter %d customer loan_number",i+1);
                scanf("%d",&c[i].loan_number);
                printf("Enter %d customer loan balance",i+1);
                scanf("%f",&c[i].balance);
        }
        for(i=0;i<n;i++)
        {
                if(c[i].balance>max)
                {
                        max=c[i].balance;
                }
        }
        for(i=0;i<n;i++)
        {
                if(c[i].balance==max)
                {
                        printf("\nHighest Loan from Bank:\nname\t loan_number\tLoan_balance");
                        printf("\n%s\t%d\t\t%f",c[i].name,c[i].loan_number,c[i].balance);

                }
        }

        getch();
}
```

9 Data Files

The input output function, Like printf( ), scanf( ), getchar( ), putchar( ), gets( ), puts( ) are known as console oriented I/O functions which always use keyboard for input device. While using these library functions, the entire data is lost when either the program is terminated or the computer is turned off. At the same time, it is cumbestsome and time consuming to handle large volume of data through keyboard. It takes a lot of time to enter the entire data. If the user makes a mistake while entering the data, he/she has to start from the beginning again. If the same data is to be entered again at some later stage, again we have to enter the same data. These problems invite the concept of data file in which data can be stored on the disks and read whenever necessary, without destroying data.

A file is a place on the disk where a group of related data is stored. The data file allows us to store information permanently and to access and alter that information whenever necessary. Programming language C has various library functions for creating and processing data files. Mainly, there are two types of data files:

1. High level (standard or stream oriented) files.
2. Low level (system oriented) files.

- **Stream-oriented** data files are generally easier to work with and are therefore more commonly used. It can be subdivided into two categories:
  - **Text files**
    A text file is a human-readable sequence of characters and the words they form that can be encoded into computer-readable formats such as ASCII. A text contains only textual characters with no special formatting such as underlining or displaying characters in boldface or different font. There is no graphical information, sound or video files. A text file known as an ASCII file and can be read by any word processor. Text file stores information in consecutive characters. These characters can be interpreted as individual data items or as a component of strings or numbers.
  - **Binary files**
    In contrast to ACII files, which contain only characters (plain text), binary files contain additional code information. A binary file is made up of machine-readable symbols that represent 1's and 0's. The binary file content must be interpreted by a program that understands in advance exactly how it is formatted. These files organize data into blocks containing contagious bytes of information. These blocks represents more complex data structures, such as arrays and structures.
- **System-oriented** data files are more closely related to the computer's operating system than stream-oriented data files. They are somewhat complicated to work with though their use may be more efficient for certain kinds of application. A separate set of procedures with accompanying library functions is required to process system oriented data files.

9.1 Opening and closing a data file:

- Before a program can write to a file or read from a file, the program must open it. Opening a file established a link between the program and the operating system. This provides the operating system, the name of the file and the mode in which the file is to be opened. While working with high level data file, we need buffer area where information is stored temporarily in the course of transferring data between computer memory and data file. The process of establishing a connection between the program and file is called opening the file.
- A structure named FILE is defined in the file **stdio.h** that contains all information about the file like name, status, buffer size, current position, and of file status, etc. All these details are hidden from the programmer and the operating system takes care of all these things.

```
typedef struct
    {
        - - - - -;
        - - - - -;
        - - - - -;
    } FILE;
```

A file pointer is a pointer to a structure of type FILE. Whenever a file is opened, a structure of type FILE is associated with it, and a file pointer that points to this structure identifies this file. The function fopen( ) is used to open a file. The buffer area is established by

FILE *ptr_variable;

And file is opened by using following syntax

ptr_variable = fopen(file_name, file_mode);

where **fopen( )** function takes two strings as arguments, the first one is the name of the file to be opened and the second one is file_mode that decides which operations (read, write, append, etc) are to be performed on the file. On success, **fopen( )** returns a pointer of type FILE and on error it returns NULL.

For example:

```
FILE *fp1, *fp2 ;
fp1 = fopen ("myfile.txt", "w");
fp2 = fopen ("yourfile.dat", "r");
```

The file opening mode specifies the way in which a file should be opened (i.e. read, write, append, etc). In other word, it specifies the purpose of opening a file. They are:

1. **"w" (write):**

   If the file dosen't exist then this mode creates a new file for writing, and if the file already exists, then the previous data is erased and the new data entered is written to the file.

2. **"a" (append):**

   If the file doesn't exist then this mode creates a new file, and if the file already exists then the new data entered is appended at the new data entered is appended at the end of existing data. In this mode, the data existing in the file is not erased as in "w" mode.

3. **"r" (read):**

   This mode is used for opening an existing file for reading purpose only. The file to be opened must exist and the previous data of file is not erased.

4. **"w+" (write + read):**

   This mode is same as "w" mode but in this mode we can also read and modify the data. If the file doesn't exist then a new file is created and if the file exists then previous data is erased.

5. **"r+" (read + write):**

   This mode is same as "r" mode but in this mode we can also write and modify existing data. The file to be opened must exist and the previous data of file is not erased. Since we can add new data and modify existing data so this mode is also called update mode.

6. **"a+" (append + read):**

   This mode is same as the "a" mode but in this mode we can also read the data stored in the file. If the file doesn't exist, a new file is created and if the file already exists then new data is appended at the end of existing data in this mode.

- The file opening modes in binary files are similar to text mode. Which are rb, rb+, wb, wb+, ab and ab+. The nature of each mode is similar to the mode of text file. Character b is added to each mode to indicate the binary mode.
- The file that was opened using **fopen( )** function must be closed when no more operations are to be performed on it. After closing the file, connection between file and program is broken.

  Its syntax is:

  fclose(ptr_variable);
- On closing the file, all the buffers associated with it are flushed and can be available for other files.

  For example:

  fclose(fp1);
  fclose(fp2);

9.2 Library functions for reading/writing from/to a file:

9.2.1 Unformatted I/O functions:

A. Character I/O functions:

fputc( ): It is used to write a character to a file.

Its syntax is

fputc(char_variable, file_ptr, vriable);

fgetc( ): It is used to read a character from a file.

Its syntax is

char_variable = fget(file_ptr_variable);

Example 9-1:

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
int main()
{
        FILE *fp;
        char ch;
        fp=fopen("first.txt","w");
        if(fp==NULL)
        {
                printf("Unable to open file");
                exit(1);
        }
        printf("Enter Character:");
        do
        {
                ch=getchar();
                fputc(ch,fp);
        }while(ch!='\n');
        fclose(fp);
        fp=fopen("first.txt","r");
        if(fp==NULL)
        {
                printf("Unable to open file");
                exit(1);
        }
        while((ch=fgetc(fp))!=EOF)
        {
                putchar(ch);
        }
        fclose(fp);
        getch();
        return 0;
}
```

B. String I/O functions:

fputs( ): It is used to write a string to a file.

Its syntax is

fputs(string, file_ptr_variable);

fgets( ): It is used to read string from file.

Its syntax is

fgets(string, int_value, file_ptr_variable);

Example 9-2:

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
#include<string.h>
int main()
{
        FILE *fp;
        char str[50];
        fp=fopen("second.txt","w");
        if(fp==NULL)
        {
                printf("Unable to open file");
                exit(1);
        }
        printf("Enter string:\n");
        while(strlen(gets(str))>0)
        {
                fputs(str,fp);
                fputs("\n",fp);
        }
        fclose(fp);
        fp=fopen("second.txt","r");
        if(fp==NULL)
        {
                printf("Unable to open file");
                exit(1);
        }
        while(fgets(str,50,fp)!=NULL)
        {
                puts(str);
        }
        fclose(fp);
        getch();
        return 0;
}
```

9.2.2 Formatted I/O functions:

fprintf( ): This function is used to write some integer, float, char or string to a file.

Its syntax is

fprintf(file_ptr_variable,"control string", list variables);

fscanf( ); This function is used to read some integer, float char or string froma file.

Its syntax is

fscanf(file_ptr_variable,"control string", & list_variables);

Example 9-3:

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
#include<string.h>
int main()
{
        FILE *fp;
        char name[50],next[4];
```

```
        int roll;
        float marks;
        fp=fopen("third.txt","w");
        if(fp==NULL)
        {
                printf("Unable to open file");
                exit(1);
        }
        do
        {
                printf("Enter name roll and marks\n");
                scanf("%s%d%f",name,&roll,&marks);
                fprintf(fp,"%s %d %f\n",name,roll,marks);
                printf("If you have next data then press yes:");
                scanf("%s",next);
        }while(strcmp(next,"yes")==0);
        fclose(fp);
        fp=fopen("third.txt","r");
        if(fp==NULL)
        {
                printf("Unable to open file");
                exit(1);
        }
        while(fscanf(fp,"%s%d%f",name,&roll,&marks)!=EOF)
        {
                printf("Name:%s\nRoll:%d\nMarks:%f\n",name,roll,marks);
        }
        fclose(fp);
        getch();
        return 0;
}
```

Output:

```
Enter name roll and marks
ram
254
75
If you have next data then press yes:yes
Enter name roll and marks
hari
256
65
If you have next data then press yes:no
Name:ram
Roll:254
Marks:75.000000
Name:hari
Roll:256
Marks:65.000000
```

9.3 End of File (EOF):

The file reading function need to know the end of file so that they can stop reading. When the end of file is reached, the opening system sends an end-of-file signal to the program. When the program receives this signal, the file reading function returns EOF, which is a constant defined in the file stdio.h.

9.4 Predefined File Pointer:

Predefined constant file pointer are opened automatically when the program starts executing and are closed when the program terminates.

File Pointer	Device
stdin	Standard input device (keyboard)
stdout	Standard O|P devices (screen)
stderr	Standard error O|P device (screen)

9.5 Record I/O in files

- Structures can be written to read from the file which is illustrate by following example:

Example 9-4:

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
#include<string.h>
int main()
{
        FILE *fp;
        int i;
        struct student
        {
                char name[50];
                float marks;
        };
        struct student s;
        fp=fopen("student.txt","w");
        if(fp==NULL)
        {
                printf("Unable to open file");
                exit(1);
        }
        for(i=0;i<20;i++)
        {
                printf("Enter name and marks\n");
                scanf("%s %f",s.name,&s.marks);
                fprintf(fp,"%s %f\n",s.name,s.marks);
        }
        fclose(fp);
        fp=fopen("student.txt","r");
        if(fp==NULL)
        {
                printf("Unable to open file");
                exit(1);
        }
        while(fscanf(fp,"%s%f",s.name,&s.marks)!=EOF)
        {
                printf("Name:%s\nMarks:%f\n",s.name,s.marks);
        }
        fclose(fp);
        getch();
        return 0;
}
```

9.6 Record I/O in binary mode

- Record I/O in example 9-4 has following problem
  - The file was in text mode. The number marks would occupy more number of bytes because each number is stored as a character string.
  - If we add more member to the structure student e.g. student.s roll no, section, age, address, phone number etc it will be very uncomfortable to write them to file using fprintf and reading them back using fscanf.
- There are function fwrite and fread for writing and reading structure (record) in a file on the disk.

fwrite( ): used for writing an entire block to a given file.

Its syntax is

fwrite(&ptr, size_of_array_or_structure, number_of_array_or_structure, fptr);

fread( ): is used to read an entire block from a given file.

Its syntax is

fread(&ptr, size_of _array_or_structure, number_of_structure_or_array, fptr);

Example 9-5: Illustration of fwrite/fread functions

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
struct employee
{
        char name[100];
        float salary;
        int age;
};
int main()
{
        struct employee emp;
        FILE *fp;
        char filename[30];
        int next;
        printf("Enter filename:");
        scanf("%s",filename);
        fp=fopen(filename,"wb");
        if(fp==NULL)
        {
                printf("Unable to open file");
                exit(1);
        }
        do
        {
                printf("Enter name:");
                scanf("%s",emp.name);
                printf("Enter salary:");
                scanf("%f",&emp.salary);
                printf("Enter age:");
                scanf("%d",&emp.age);
                fwrite(&emp,sizeof(emp),1,fp);
                printf("Enter 1 to continue to read next data set:");
                scanf("%d",&next);
        }while(next==1);
        fclose(fp);
        fp=fopen(filename,"rb");
        if(fp==NULL)
        {
```

```
            printf("Unable to open file");
            exit(1);
        }
        printf("\nData Stored in File:\n");
        while(fread(&emp,sizeof(emp),1,fp)==1)
        {
            printf("Name:%s\tSalary:%f\tAge:%d\n",emp.name,emp.salary,emp.age);
        }
        fclose(fp);
        getch();
        return 0;
}
```

Output:

```
Enter filename:employee
Enter name:ram
Enter salary:25000
Enter age:22
Enter 1 to continue to read next data set:1
Enter name:hari
Enter salary:30000
Enter age:25
Enter 1 to continue to read next data set:0

Data Stored in File:
Name:ram        Salary:25000.000000     Age:22
Name:hari       Salary:30000.000000     Age:25
```

9.7 Random Access to File

We can access the data stored in the file in two ways, sequentially or random. So, far we have used only sequentially access in our programs. For example, if we want to access the forty-fourth record then first forty three records should be read sequentially to reach the forty-fourth record. In random access, data can be accessed and processed randomly i.e. in this case the forty-forth record can be accessed directly. There is no need to read each record sequentially, if we want to access a particularly record. Random access takes less time than the sequential access.

C supports these functions for random access file processing:

- **fseek( )**
- **ftell( )**
- **rewind( )**

fseek( ): This function is used for setting the file position pointer at the specified byte. The syntax is

fseek(fp, displacement, origin)

where fp is file pointer, displacement is long integer which can be positive or negative and it denotes the number of bytes which are skipped backward (if negative) or forward (if positive) from the position specified in the third argument.

The third argument named origin is the position relative to which the displacement takes place. It can take one of these three values:

Constant	Value	Position
SEEK_SET	0	Beginning of file
SEEK_CUR	1	Current position
SEEK_END	2	End of File

Example:

```
fseek(fp,0,SEEK_SET)        // Moves file pointer at the beginning of file
fseek(fp,0,SEEK_END)        //Moves file pointer at the end of the file
fseek(fp,10, SEEK_SET)      // Moves the file pointer at 10 bytes right from the beginning of the file
fseek(fp,-2,SEEK_CUR)       //Moves the file pointer at 2 bytes left from current position
```

rewind( ): This function places the pointer to the beginning of the file, irrespective of where it is present right now.

rewind(fp);

ftell( ): This function returns the current position of the file position. The value is counted from the beginning of the file. The syntax is

position= ftell(fp); Where position is a long int

Example 9-6:

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
struct employee
{
        char name[100];
        float salary;
        int age;
};
int main()
{
        struct employee emp;
        FILE *fp;
        char filename[30];
        int next;
        printf("Enter filename:");
        scanf("%s",filename);
        fp=fopen(filename,"wb+");
        if(fp==NULL)
        {
                printf("Unable to open file");
                exit(1);
        }
        do
        {
                printf("Enter name:");
                scanf("%s",emp.name);
                printf("Enter salary:");
                scanf("%f",&emp.salary);
                printf("Enter age:");
                scanf("%d",&emp.age);
                fwrite(&emp,sizeof(emp),1,fp);
                printf("Enter 1 to continue to read next data set:");
```

```
            scanf("%d",&next);
        }while(next==1);
        rewind(fp);
        if(fp==NULL)
        {
            printf("Unable to open file");
            exit(1);
        }
        printf("\nData Stored in File:\n");
        while(fread(&emp,sizeof(emp),1,fp)==1)
        {
            printf("Name:%s\tSalary:%f\tAge:%d\n",emp.name,emp.salary,emp.age);
        }
        fclose(fp);
        getch();
        return 0;
    }
```

Output:

```
Enter filename:employee
Enter name:ram
Enter salary:20000
Enter age:21
Enter 1 to continue to read next data set:1
Enter name:hari
Enter salary:30000
Enter age:25
Enter 1 to continue to read next data set:0

Data Stored in File:
Name:ram        Salary:20000.000000     Age:21
Name:hari       Salary:30000.000000     Age:25
```

Some Solved Example

1. **Write a program to read name and roll number of 25 students from the user and store them in a file. If the file already contains data, your program should add new data at the end of the file.**

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
int main()
{
        char name[50];
        int roll,i;
        FILE *fp;
        fp=fopen("student.txt","a");
        if(fp==NULL)
        {
                printf("Unable to open file");
                exit(1);
        }
        for(i=0;i<25;i++)
        {
                printf("Enter Name:");
                scanf("%s",name);
                printf("Enter roll:");
                scanf("%d",&roll);
                fprintf(fp,"%s %d\n",name,roll);
        }
        fclose(fp);
        getch();
        return 0;
}
```

2. **Write a program to create a file "Employee.dat" having several data of the following format of the following Employee code, Employee name, date of birth and salary.**

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
struct employee
{
        char code[10];
        char name[50];
        int dob;
        float salary;
};
main()
{
        struct employee e[50];
        int n,i;
        FILE *fp;
        fp=fopen("e://employee.dat","w");
        printf("enter number of employee:");
        scanf("%d",&n);
        for(i=0;i<n;i++)
        {
                printf("Enter detail of employee %d\n",i+1);
                printf("enter employee code:");
```

```
                scanf("%s",&e[i].code);
                printf("enter employee name:");
                scanf("%s",&e[i].name);
                printf("enter employee date of birth:");
                scanf("%d",&e[i].dob);
                printf("enter employee salary:");
                scanf("%f",&e[i].salary);
                /* write to the file */
                fprintf(fp,"%s %s %d %f\n",e[i].code,e[i].name,e[i].dob,e[i].salary);
        }
        /* close the file */
        fclose(fp);
        getch();
}
```

3. Write another program to access the above file and print :

-Employee details who were born before year 1980.

-Average salary.

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
struct employee
{
        char code[10];
        char name[50];
        int dob;
        float salary;
};
main()
{
        struct employee e[50];
        float sum=0,avg;
        int i=0;
        FILE *fp;
        fp=fopen("e://employee.dat","r");
         if (fp == NULL)
        {
                printf("I couldn't open employee.dat\n");
                exit(0);
         }

        printf("code\tName\tDOB\tsalary\n");
        while (fscanf(fp,"%s%s%d%f\n",&e[i].code,&e[i].name,&e[i].dob,&e[i].salary)!=EOF)
        {
                sum=sum+e[i].salary;
                if(e[i].dob<1980)
                        printf("%s\t%s\t%d\t%f\n",e[i].code,e[i].name,e[i].dob,e[i].salary);
                i++;
         }
        avg=sum/i;
        printf("\nAverage salary=%f",avg);
        /* close the file */
        fclose(fp);
        getch();
}
```

4. Write a program to read name, roll and marks obtained by students in five subjects and store the result in file until the user is not satisfied. The program should have another portion to read all information. Your program should ask the user to read from or write to the file and call the specific functions.

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
void write_to_file();
void read_from_file();
struct student
{
        char name[50];
        int roll;
        float marks[5];
};
int main()
{
        int n;
        printf("Enter\n1. for writing to file\n2. for reading from file\n");
        scanf("%d",&n);
        switch(n)
        {
                case 1:
                        write_to_file();
                        break;
                case 2:
                        read_from_file();
                        break;
                default:
                        printf("Enter 1 or 2");
        }
        getch();
        return 0;
}
void write_to_file()
{
        FILE *fp;
        struct student s;
        int i;
        char next;
        fp=fopen("student.dat","wb");
        if(fp==NULL)
        {
                printf("Unable to open file");
                exit(1);
        }
        do
        {
                printf("\nEnter name:");
                scanf("%s",s.name);
                printf("Enter Roll:");
                scanf("%d",&s.roll);
                for(i=0;i<5;i++)
                {
                printf("Enter marks of subject %d:",i+1);
```

```
                scanf("%f",&s.marks[i]);
                }
                fwrite(&s,sizeof(s),1,fp);
                printf("Enter y for next set of data else any key:");
                next=getche();
        }while(next=='y');

        fclose(fp);
}

void read_from_file()
{
        FILE *fp;
        struct student s;
        int i;
        fp=fopen("student.dat","rb");
        if(fp==NULL)
        {
                printf("Unable to open file");
                exit(1);
        }
        printf("Data Stored in file\n");
        while(fread(&s,sizeof(s),1,fp)==1)
        {
                printf("Name:%s\nRoll:%d\n",s.name,s.roll);
                for(i=0;i<5;i++)
                {
                        printf("Marks %d=%f\n",i+1,s.marks[i]);
                }
        }

        fclose(fp);
}
```