UNIT-I

Introductory Concepts: Introduction to computers, What is a Computer, Block diagram of Computer, Computer Characteristics, Hardware Vs Software, How to develop a program, Software development life cycle, Structured programming, Modes of operation, Types of programming languages, Introduction to C, Desirable program characteristics.

Introduction to Computer problem solving: Introduction, The problem solving aspect, Top down design, Implementation of algorithms.

Introduction to C programming: The C character set, Writing first program of C, Identifiers and key words, A more useful C program, Entering the program into the computer, Compiling and executing the program, Data types, Constants, Variables and arrays, Declarations, Expressions, Statements, Symbolic Constants.

Operators and Expressions: Arithmetic operators, Unary operators, Relational and Logical operators, Assignment operators, Conditional operator, Library functions.

Fundamental algorithms: Exchanging the values of two variables, Factorial computation, Sine function computation, Reversing the digits of an integer, Generating prime numbers

INTRODUCTION TO COMPUTERS

Now a day's every aspect of our life depends on Computer. Computer is a machine that can receive, store, transform, and output the data of different kinds (i.e., numbers, text, images, sound etc.)

The computer program's role is essential in this technology; without a list of instructions to follow, the computer is virtually useless. Programming languages allows us to write these programs and through that we can communicate with computers.

The First electronic computer was built in the late 1930s by **Dr. John Atanasoff** and **Clifford Berry** at Iowa State University in U.S.A.

The first large-scale, general purpose electronic digital computer called the ENIAC(ELECTRONIC NUMERIC INTEGRATOR AND COMPUTER), weighing 30 tons and occupying a 30 X 50 foot space.

Modern computers are categorized according to their size and performance.

The largest capacity and fastest mainframes are called ***Super computers*** and are used by research laboratories and in computationally intensive applications such as weather forecasting etc.

Large real – time transaction processing systems, such as ATMs and other banking networks, and corporate reservation systems for hotels, airlines uses ***mainframe*** **computers**.

Personnel computers are use by a single person at a time. Personal computers are small and inexpensive. In fact, portable, battery-powered "laptop" computers weighing less than ***5*** or 6 pounds are now widely used by many students and traveling professionals. Personal computers are used extensively in most schools and businesses and they are rapidly becoming common household items.

Many organizations connect personal computers to larger computers or to other personal computers, thus permitting their use either as stand-alone devices or as terminals within a computer ***network.*** Connections over telephone lines are also common.

WHAT IS COMPUTER?

A computer is an electronic device that can store, retrieve and process data at speed millions or billions times faster than a human.

It can be used for various purposes, which is possible by using programs for different applications.

Program is set of instructions written in a particular sequence in computer related language.

CHARACTERISTICS OF COMPUTERS

Speed

Computer can perform millions (1,000,000) of instructions and even more per second. Therefore, we determine the speed of computer in terms of microsecond (10-6 part of a second) or nano-second (10-9 part of a second). From this you can imagine how fast your computer performs work.

Accuracy

In addition to speed, the computer should have accuracy or correctness in computing. The degree of accuracy of computer is very high and every calculation is performed with the same accuracy. The accuracy level is determined on the basis of design of computer. The errors in computer are due to human and inaccurate data.

Versatility

It means the capacity to perform completely different type of work. You may use your computer to prepare payroll slips. Next moment you may use it for inventory management or to prepare electric bills.

Power of Remembering

Computer has the power of storing any amount of information or data. Any information can be stored and recalled as long as you require it, for any numbers of years. It depends entirely upon you how much data you want to store in a computer and when to lose or retrieve these data.

No IQ

Computer is a dumb machine and it cannot do any work without instruction from the user. It performs the instructions at tremendous speed and with accuracy. It is you to decide what you want to do and in what sequence. So a computer cannot take its own decision as you can.

No Feeling

It does not have feelings or emotion, taste, knowledge and experience. Thus it does not get tired even after long hours of work. It does not distinguish between users.

Storage

The Computer has an in-built memory where it can store a large amount of data. You can also store data in secondary storage devices such as floppies, which can be kept outside your computer and can be carried to other computers.

HARDWARE vs SOFTWARE

The elements of computer system fall into two major categories:

- **HARDWARE**
- **SOFTWARE**

COMPUTER HARDWARE:

Hard ware is the equipment used to perform the necessary computations.

Basic Computer Hardware Components:

- Main memory
- Central Processing Unit (CPU)
- **Input Units :** Keyboard, Mouse, Scanner
- **Output Units:** Monitor, Printer, Plotter, Scanner, Modem, Speaker

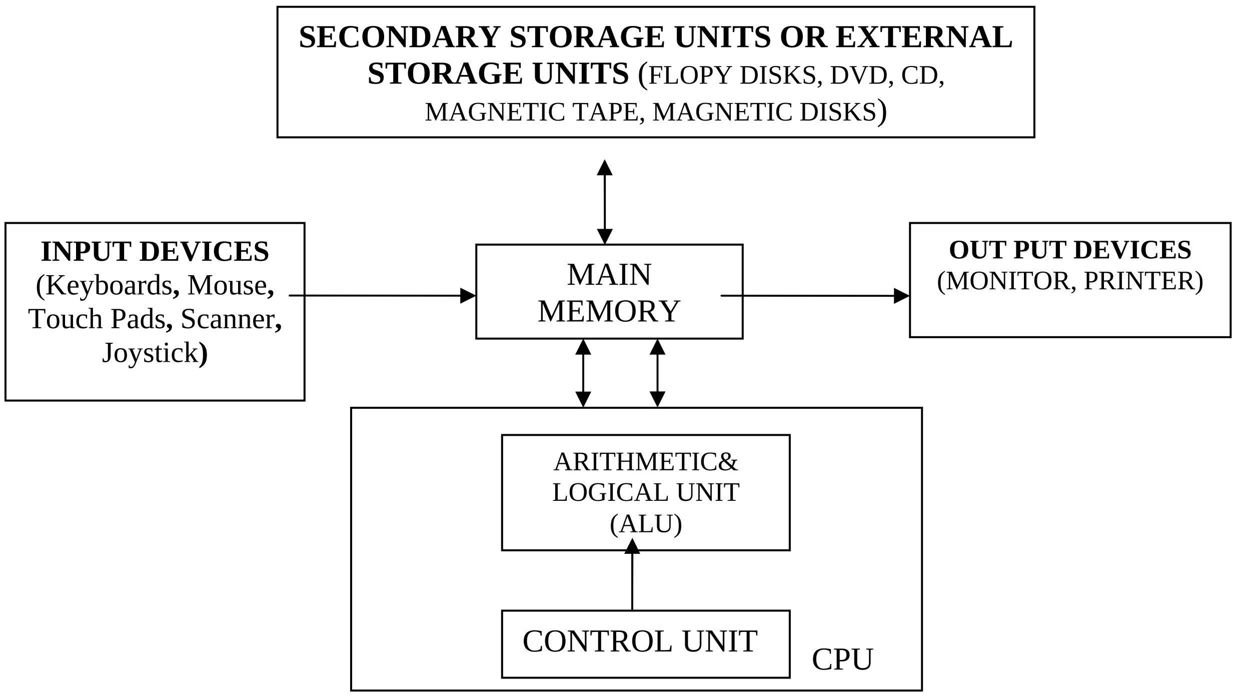

SECONDARY STORAGE UNITS OR EXTERNAL STORAGE UNITS (FLOPY DISKS, DVD, CD, MAGNETIC TAPE, MAGNETIC DISKS)
INPUT DEVICES (Keyboards, Mouse, Touch Pads, Scanner, Joystick)
MAIN MEMORY
OUT PUT DEVICES (MONITOR, PRINTER)
ARITHMETIC& LOGICAL UNIT (ALU)
CONTROL UNIT
CPU

Fig: Block diagram for the hardware parts of a digital computer

The program must first be translated from secondary storage to main memory before it can be executed. Normally the person using a program (the program user) must supply data to be processed. These data are entered through an input device and can be accessed and manipulated by CPU. The results of this manipulation are then stored back in main memory.

Finally, the information in the main memory can be displayed through an output device.

MEMORY

It is an essential component in any computer. Consists of memory cells, which is consist of ordered sequence of storage locations.

Address of a memory cell: the relative position of a memory cell in the computer's main memory.

Contents of a memory cell: The information stored in a memory cell, either a program instruction or data.

Bits and Bytes: Binary refers to a number system based on two numbers, 0 and 1, so a bit is either a 0 or a 1. Generally there are <u>*eight bits to a byte.*</u>

Storage and Retrieval of Information in Memory: A computer can either store a value into a memory in terms of binary number representations or retrieve a value from the memory.

Computers use two types of storage locations, one for storing the data that are being currently handled by CPU and the other for storing the results and future use. The storage location where the data are held temporarily is referred as the primary memory while the storage location where the program and data are stored permanently for future use is referred to as the secondary memory.

The data and instructions stored in the primary memory can be directly accessed by the CPU by using data and address buses. However the data stored in the secondary memory is not directly access by the CPU. First the data is transferred to primary memory and then to the CPU.

Memory of a computer is divided into two types

1) Primary Memory or Storage (or Main Memory)

2) Secondary Memory or Storage.

- **MAIN MEMORY: Main** memory stores programs, data and results. Most computers have two types of main memory.

Random-Access-Memory (RAM) : Used for temporary storage of programs and data. Everything in RAM will be lost when the computer is switch is off (i.e, volatile memory). All kinds of processing of the CPU are done in this memory. RAM is a semi-conductor memory and is always faster than magnetic memories.

Read-Only-Memory (ROM): Used for permanently storage of programs and data. The computer can retrieve, but cannot store, information in ROM, hence its name, read-only. The data stored in ROM stay permanently even after the power is switched off and therefore ROM is a non-volatile memory. Start up instructions and critical instructions are burned into ROM chips at factory. ROM is needed in the computer to store the Basic Input Output System (BIOS). BIOS is responsible to make the computer ready for working by the user. This process is called BOOTING.

- **SECONDARY STORAGE DEVICES:** These devices are used to store huge amount of information. These are Hard disk, CDs, DVDs and flash memories.

Secondary Memory Characteristics:

• Connected to main memory.

• Used to hold programs and data.

• The content of secondary storage is easily changed.

• Used for long-term storage.

• Huge capacity compared to main memory, but access is slow.

• Usually data and programs are organized into files in secondary memory.

Hard Disk: Hard disks are attached to their disk drives and coated with a magnetic material**.** Each data bit is a magnetized spot on the disc, and the spots are arranged in concentric circles called tracks. The disc drive read/write head accesses data by moving across the spinning disk to the correct track and then sensing the spots as they move by. Hard discs hold from one to several hundred gigabytes (GB) of data, but clusters of hard drives that stores data from an entire network may provide as much as a terabyte (TB) of storage.

Terms used to quantify storage capacities:

Term	Abbreviation	Equivalent to	Comparison to Power of 10
Byte	B	8 Bits	
KiloByte	KB	1,024 (bytes)	>10**3
MegaByte	MB	1,048,576(bytes)	>10**6
GigaByte	GB	1,073,751,824(bytes)	>10**9
TeraByte	TB	1,099,511,627,776(bytes)	>10**12

Optical Drives: For storing and retrieving data on compact discs(CDs) or digital versatile disks(DVDs) that can be removed from the drive. One CD can hold 680 MB of data. A DVD uses smaller pits packed in a tighter spiral, allowing storage of 4.7 GB of data one one layer. Some DVDs can hold four layers of data- two on each side for a total capacity of 17 GB, sufficient storage for as much as nine hours of studio quality video and multi-channel audio.

Flash Drive (USB)(Universal Serial Bus):Flash drives(stick memory) have no moving parts and all data transfer is by electronic signal only. Typical USB flash drives store 1 to a few GB of data, but 64 GB drives are also available.

Information stored on a disk is organized into separate collections called **files**: One file may contain a C program. Another file may contain the data to be processed by the program(a data file) or a file may contain a picture information. A file could contain results generated by a program(an output file).

CENTRAL PROCESSING UNIT (CPU):

The Central Processing Unit (CPU) has two roles.

A) Coordinating Operations of Computer

B) Performing Arithmetic and Logical operations on data.

The CPU follows the instructions contained in a computer program to determine which operations should be carried out and in what order. The CPU fetches an instruction, interprets the instruction to determine what should be done and then retrieves any data needed to carry out that instruction .The CPU stores the results in main memory. The CPU can perform such arithmetic operations as addition, subtraction, multiplication, and division. The CPU can also compare the contents of two memory cells and make decision based on the result of that comparison.

A CPU's current instruction and data values are stored temporarily inside the CPU in special high-speed memory locations called **registers (high speed memory location).**

INPUT/OUTPUT DEVICES

We use Input/Output devices to communicate with the computer. They allow us to enter data for a computation and to observe the results of that computation.

Input/output device, also known as computer peripheral, any of various devices used to enter information and instructions into a computer for storage or processing and to deliver the processed data to a human operator.

Following are few of the important input devices which are used in Computer Systems

- Keyboard
- Mouse
- Joy Stick
- Light pen
- Track Ball
- Scanner
- Graphic Tablet
- Microphone
- Magnetic Ink Card Reader(MICR)
- Optical Character Reader(OCR)
- Bar Code Reader
- Optical Mark Reader

Mouse

Mouse is most popular pointing device. It is a small palm size box with a round ball at its base which senses the movement of mouse. A *mouse* is an input device used to select elements on the screen, such as tools, icons and buttons, by pointing and clicking them .We can also use a mouse to draw and paint on the screen of the computer system.

Generally it has two buttons called left and right button and scroll bar is present at the mid. Mouse can be used to control the position of cursor on screen.

Advantages

- Easy to use
- Not very expensive
- Moves the cursor faster than the arrow keys of keyboard.

Keyboard

We use a *keyboard* as an input device to give input to computer. A keyboard has keys for letters, numbers, and punctuation marks plus some extra keys for performing special functions

Keyboard is of two sizes 84 keys or 101/102 keys, but now 104 keys or 108 keys keyboard is also available.

The keys are following

Sr. No.	Keys	Description
1	Typing Keys	These keys include the letter keys (A-Z) and digits keys (0-9).
2	Numeric Keypad	It is used to enter numeric data or cursor movement.
3	Function Keys	The twelve functions keys are present on the keyboard. These are arranged in a row along the top of the keyboard.Each function key has unique meaning and is used for some specific purpose.
4	Control keys	These keys provides cursor and screen control. It includes four directional arrow key.Control keys also include Home, End,Insert, Delete, Page Up, Page Down, Control(Ctrl), Alternate(Alt), Escape(Esc).
5	Special Purpose Keys	Keyboard also contains some special purpose keys such as Enter, Shift, Caps Lock, Num Lock, Space bar, Tab.

Joystick

Joystick is also a pointing device which is used to move cursor position on a monitor screen. It is a stick having a spherical ball at its both lower and upper ends. The Joystick can be moved in all four directions.

The function of joystick is similar to that of a mouse. It is mainly used in Computer Aided Designing (CAD) and playing computer games.

Light Pen

Light pen is a pointing device which is similar to a pen. It is used to select a displayed menu item or draw pictures on the monitor screen. It consists of a photocell and an optical system placed in a small tube.

Track Ball

Track ball is an input device that is mostly used in notebook or laptop computer, instead of a mouse. This is a ball which is half inserted and by moving fingers on ball, pointer can be moved.

Since the whole device is not moved, a track ball requires less space than a mouse. A track ball comes in various shapes like a ball, a button and a square.

Scanner

Scanner is an input device which works more like a photocopy machine. It is used when some information is available on a paper and it is to be transferred to the hard disc of the computer for further manipulation.

Microphone

Microphone is an input device to input sound that is then stored in digital form. The microphone is used for various applications like adding sound to a multimedia presentation or for mixing music.

Magnetic Ink Card Reader (MICR)

MICR input device is generally used in banks because of a large number of cheques to be processed every day. The bank's code number and cheque number are printed on the cheques with a special type of ink that contains particles of magnetic material that are machine readable.

Bar Code Readers

Bar Code Reader is a device used for reading bar coded data (data in form of light and dark lines). Bar coded data is generally used in labeling goods, numbering the books etc. It may be a hand held scanner or may be embedded in a stationary scanner.

Bar Code Reader scans a bar code image, converts it into an alphanumeric value which is then fed to the computer to which bar code reader is connected.

Optical Mark Reader (OMR)

OMR is a special type of optical scanner used to recognize the type of mark made by pen or pencil. It is specially used for checking the answer sheets of examinations having multiple choice questions.

Following are few of the important output devices which are used in Computer Systems

- Monitors
- Graphic Plotter
- Printer

Monitors

Monitor commonly called as Visual Display Unit (VDU) is the main output device of a computer. It forms images from tiny dots, called pixels that are arranged in a rectangular form. The sharpness of the image depends upon the no. of the pixels.

There are two kinds of viewing screen used for monitors.

- Cathode-Ray Tube (CRT)
- Flat- Panel Display

Cathode-Ray Tube (CRT) Monitor

In the CRT display is made up of small picture elements called pixels for short. The smaller the pixels, the better the image clarity, or resolution. It takes more than one illuminated pixel to form whole character, such as the letter e in the word help.

Flat-Panel Display Monitor

The flat-panel display refers to a class of video devices that have reduced volume, weight and power requirement compare to the CRT. You can hang them on walls or wear them on your wrists. Current uses for flat-panel displays include calculators, videogames, monitors, laptop.

Printers

Printer is the most important output device, which is used to print information on paper.

Dot Matrix Printer

In the market one of the most popular printer is Dot Matrix Printer because of their ease of printing features and economical price. .

Advantages

- Inexpensive
- Widely Used
- Other language characters can be printed

Disadvantages

- Slow Speed
- Poor Quality

Laser Printers

These are non-impact page printers. They use laser lights to produces the dots needed to form the characters to be printed on a page.

Advantages

- Very high speed.
- Very high quality output.
- Give good graphics quality.
- Support many fonts and different character size.

Disadvantage

- Expensive.
- Cannot be used to produce multiple copies of a document in a single printing.

Speaker: It is an electromechanical device that converts an electrical signal into sound. These provide audio output.

COMPUTER SOFTWARE

Software: It is a set of computer programs that are required to enable the hardware components to work and perform their respective operations effectively.

Computer Program: It is a set of logical instructions, written in computer programming language that tells the computer how to execute a particular task.

The software is divided into two types **1) System Software** **2) application Software**

1) System Software: It consists of many different programs that manage and support different tasks. Depending upon the task performed, the system software can be classified into two major groups:

a) System management programs: Used for managing both the hardware and software systems.

System management program include:

- Operating system(Ex: MS DOS,WINDOWS,LINUX)
- Utility programs(Ex: Virus scanner)
- Device drivers(Ex: I/O device driver, printer driver)

b) System Development Programs: These are known as programming software allows the user to develop programs in different programming languages. This includes:

- Language Translators: translate the code form one language to another
- Linkers: Link the various modules and objects with the library files
- Debuggers: help in debugging the program(i.e. identifying the errors)
- Editors: Help to write the program code

OPERATING SYSTEM (OS): The collection of computer programs that control interaction of user and computer hardware is called Operating System. Most of the operating system are written using C language and has a lot of sub modules in it. OS manages allocation of computer resources. Usually part of the OS is stored permanently in a read-only memory (ROM) chip work as a starter of the operating system is called booting the computer.

The following is a list of some of the operating System’s capabilities.

- Communicating with the computer user.
- Managing allocation of memory, of processor time, and of other resources for various task.
- Managing Input/output operations among the user programs and OS.
- Accessing data from secondary storage &Writing data to secondary storage.

Command-Line Interface	Graphical User Interface
UNIX	Macintosh OS
MS-DOS	Windows
VMS	UNIX + X Window System

Widely Used Operating System Families Categorized by User Interface Type

System Development Tools: The Successful development and execution of programs requires the usage of a number of tools. Some of these tools are:

Language translators, Linkers, Debuggers, Editors

1) Language translators:

- **Assembler:** An assembler is a computer program that translates assembly language statements into machine language codes. The assembler takes each of the assembly language statements from the source code and generates a corresponding bit stream contains 1's and 0's.The output of the assembler in the form of sequence of 1's and 0's is called *Object code* or *machine code*. This machine code is finally executed by CPU to obtain results.
- **Compiler:** The compiler is a computer program that translates the source code written in a high-level language into *object code* of the machine level or low-level language. This translation process is called compilation. The entire high-level program is converted into the executable machine code file. Examples for compiled languages are: COBOL, FORTRAN, C, C++ etc.

Compilers are classified as two types: 1) single-pass compilers and ii) multi-pass compilers. Though Single pass compilers are generally faster than multi-pass compliers, for sophisticated optimization, multi-pass assemblers are required generate high-quality code.

- **Interpreter:** The interpreter is a translation program that converts each high-level program statement into the corresponding machine code. This translation process carried out just before the program statement is executed. Instead of the entire program, one statement at a time is translated and executed immediately.

Ex: BASIC and PERL are interpreted languages.

Difference between compiler and interpreter:

1) Compiler translates the entire source code of high-level language program into machine code but interpreter translates only one statement of source code into machine-level language.

2) The compiled languages can be executed more efficiently and are faster than interpreter.

2) Linkers: Most of the High-level languages allow the developer to develop a large program containing multiple modules. Linker arranges or combines the object code of all the modules that have been generated by the language translator into a single executable program. The CPU of the computer is incapable of linking all the modules at the execution time and therefore, linker is combining all the modules into a single program. Linker also includes the links of various objects, which are defined in the runtime libraries.

The following figure shows the working of a linker.

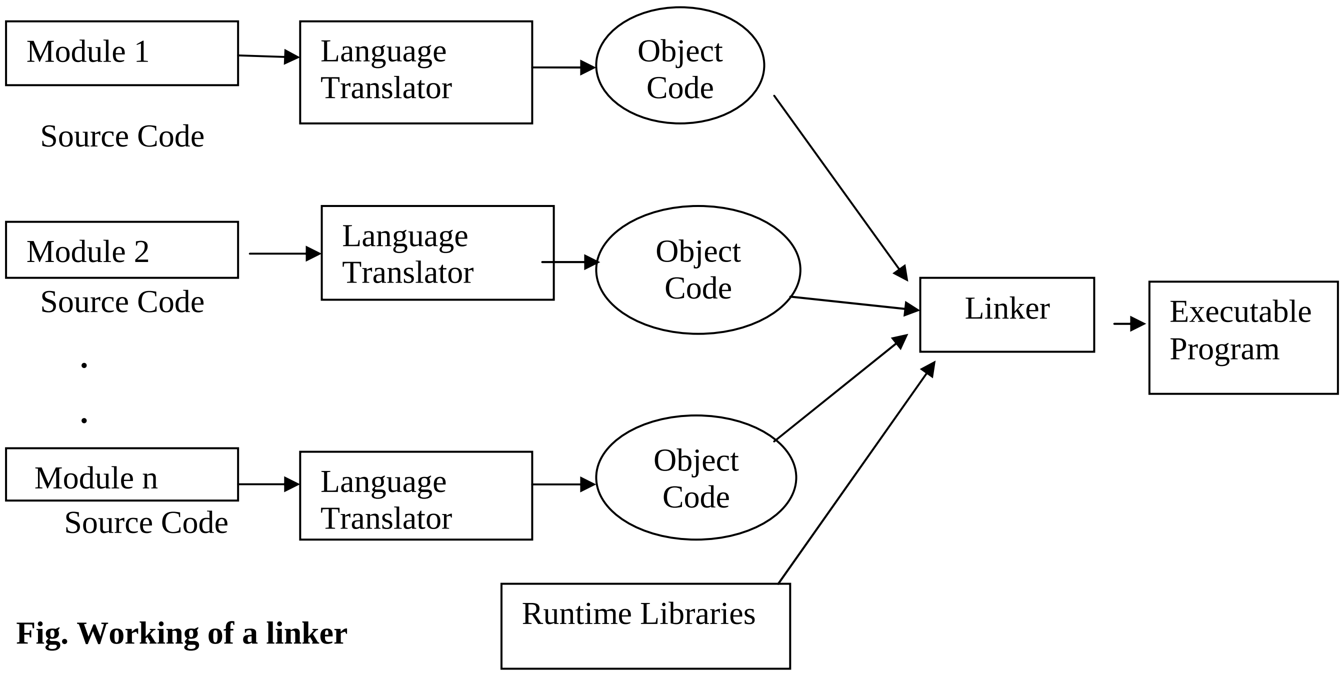


Fig. Working of a linker

3) Debugger: Debugger is the software that is used to detect the errors and bugs present in the programs. The Debugger locates the position of the errors in the program code with the help of Instructor Set Simulator (ISS) technique. ISS is capable of stopping the execution of a program at the point where an erroneous statement is encountered.

Debugger is divided into two types: i) Machine-level debugger and ii) symbolic debugger.

Machine-level debugger debugs the **object code** of the program and shows all the lines where bugs are detected.

The symbolic debugger debugs the **original code**. i.e., the high-level language code of the program. It shows the position of the bug in the original code of the program developed by the programmer.

The debugger performs a number of functions other than debugging, such as inserting breakpoints in the original code, tracking the value of specific variable, etc.

4) Editors: Editor is a special program that allows the user to work with text in a computer system. It is used for the documentation purpose and enables us to edit the information present in an existing document or a file. The editor enables us to perform various editing operations such as copy, cut and paste while editing the text. The editors are divided into following categories:

- Text editor: It is used to edit plain text.
- Graphics editor: It is used to edit the information related to graphical objects.
- Binary editor: It is used to edit the digital data or binary data (1’s and 0’s).
- HTML editor: It is used to edit the information included in the web pages.
- Source Code editor: It is used to edit the source code of a program written in a programming language such as C, C++ and java.

2) APPLICATION SOFTWARE: It is performing any task or solving a particular problem.

This is further divided into two types:

a. ***Packages*****:** Collection related Applications are called "Packages".

b. ***Languages***: It is a systematical code for communication between two persons.

For example, a word-processing application MS Word, WordPerfect, or a spreadsheet application or Excel programs or a database management application such as Access or dBASE systems are well known application programs. User can also create their own application programs by using any programming language and solve a specific problem.

COMPUTER LANGUAGES:

It is a systematical code for communication between System and user. This is in two categories.

1) Low Level Language: Machine dependable language is called "*Low level language*". This is further divided into two types.

a. ***Machine language*****:** Machine understandable language is called "Machine language". It is understandable by BINARY LANGUAGE. It is in the form of 0's and 1's.

Advantages:

i. Computer can understand directly.

Disadvantages:

i. It is very difficult to remember the codes and address of memory locations.

ii. User can’t modify the program.

iii. User can’t debug the program.

iv. It is machine dependent.

b. ***Assembly language****.* : It is a code language. It has some codes for performing specific operations. This is better than machine language.

E.g. ADD, SUB, MUL, DIV, ABS, FACT etc.

Advantages:

i. User can remember the mnemonics.

ii. It is easy to understand and develop the programs.

iii. User can modify the program and debug.

iv. It is suitable for simple applications.

Disadvantages:

i. It is machine dependent.

ii. It requires the translator program called Assembler

2) **High Level Language**: machine independent programming language that combines algebraic expressions and English symbols User understandable language is called "*High level language*".

Advantages:

i. Easy to understand
ii. Easy to modify and debug.
iii. Suitable for complex applications.

Disadvantages:

i. It requires the translator program called Compiler or Interpreter.
ii. It runs programs slower with compare to low level languages

To solve a given problem using computer, user prefer high-level programming languages. There are many high-level programming languages are available these days.

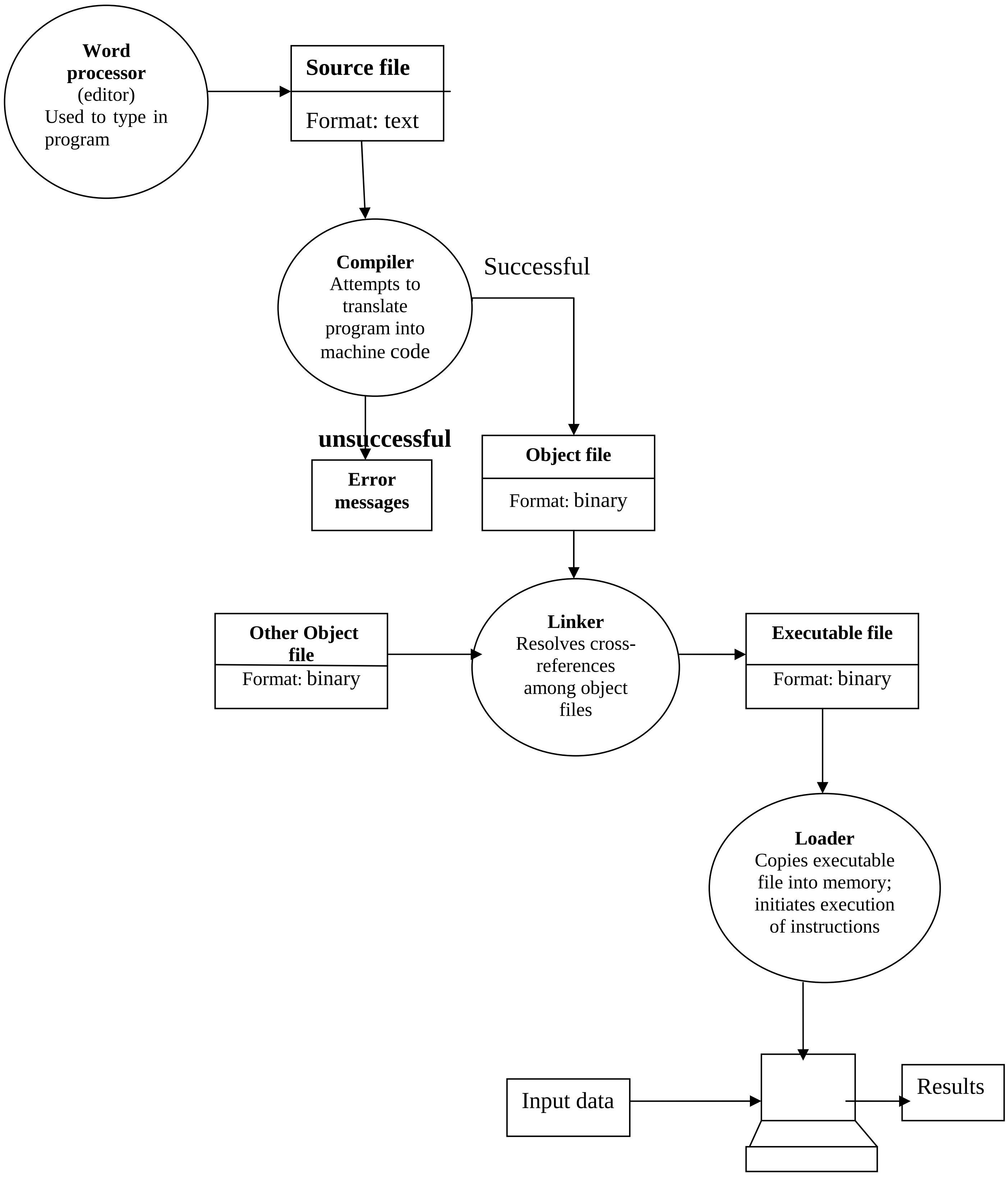
Language	Application Area	Origin of Name
FORTRAN	Scientific	FORmula TRANslation
COBOL	Business data processing	**Co**mmon **B**usiness **O**riented **L**anguage
LISP	Artificial Intelligence	**Lis**t **P**rocessing
C	System programming	Predecessor language was named B
Prolog	Artificial Intelligence	**Log**ic **Pro**gramming
Ada	Real-Time distributed sys.	Ada Augusta Byron Collaborated..
Smalltalk	GUI	Objects "talk" to one another with message
C++	Support OOP	Incremental modification of C
Java	Supports web programming	originally named "oak"

Source file is written by a programmer for the solution of a specific problem. A **source** files containing the text of a high-level language program. The software developer creates this file by using an editor or word processor.

The high-level language before it is to be executed, it must first be translated into a computer's machine language.The program that do this translation is called **COMPILER.**
During compilation grammar rules are checked.(**syntax checking**) If the program is syntactically correct, compiler saves in an object file . Object file consist of machine language instructions. Using **linker** program object file is converted into machine language code which is an executable file and ready to run.

As the program executes it takes input data from one or more sources and sends results to output and /or secondary storage devices.

Entering, translating and running a high level language program:


Word processor (editor) Used to type in program
Source file
Format: text
Compiler Attempts to translate program into machine code
Successful
unsuccessful
Error messages
Object file
Format: binary
Other Object file
Format: binary
Linker Resolves cross-references among object files
Executable file
Format: binary
Loader Copies executable file into memory; initiates execution of instructions
Input data
Results

HOW TO DEVELOP A PROGRAM:

To write a program for any task we need to identify the steps and their execution sequence. Every program consists of mainly three parts input, processing and output. The starting point for solving any problem is to identify its input and output.

For example, we want to write a program to add two numbers; first we need to identify the inputs from the given problem statement. Here the inputs are the two numbers to be added. Next step is to identify the output. The result of addition of the two numbers will be the output for the problem. The conversion of input to output defines the steps for the processing.

We require three steps:

1. Read /Input two numbers
2. ADD these two numbers
3. Display/Output the result of addition

These steps are written in English language and are called as ALGORITHM. When an algorithm is translated using a specific high level language, then it becomes a program.

ALGORITHM

Algorithm is a very popular technique used to obtain solution for the given problem. An Algorithm is defined as '*a finite set of steps that provide a chain of actions for solving a definite nature of problem'*.

An algorithm is a method of representing the step-by-step procedure for solving a problem. An algorithm is useful for finding the right answer to a problem or to a difficult problem by breaking the problem into simple cases.

An algorithm must possess the following properties:

i. **Finiteness:** An algorithm should terminate in a finite number of steps.

ii. **Definiteness:** Each step of the algorithm must be precisely stated (unambiguous).

iii. **Effectiveness:** Each step must be effective, in the sense that it should be easily convertible into program statement and can be performed exactly in a finite amount of time.

iv. **Generality:** The algorithm should be complete in itself so that it can be used to solve all problems of a given type for any input data.

v. **Input/output:** Each algorithm must take zero, one or more quantities as input data and yield one or more output values.

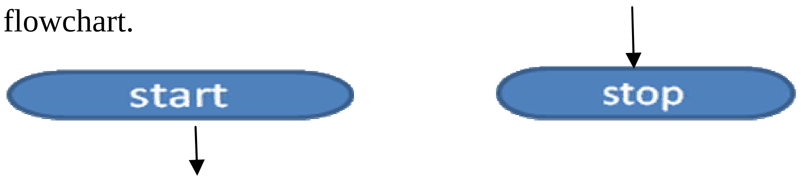
PSEUDO CODE:

Another useful representation of the detailed step is pseudo code. It is a more formal representation than the algorithm. In Pseudo code representation each of the step will be written via operators and statements equivalent to some programming language instructions. The only difference will be that the exact syntax of the programming language will not be followed. All pseudo code will start with key word "START" and complete with keyword "END" or "STOP".

FLOW CHART

- ✓ A flow chart is a visual representation of the sequence of steps for solving a problem.
- ✓ A completed flow chart is an alternative technique for solving a problem
- ✓ A flow chart is a set of symbols that indicate various operations in the program
- ✓ For every process, there is a corresponding symbol in the flowchart. All symbols are interconnected by arrows to indicate the flow of information and processing.
- ✓ Once an algorithm is written its pictorial representation can be done using flowchart symbols.

- **Start and end**: The start and end symbol indicate both the beginning and the end of the flowchart.

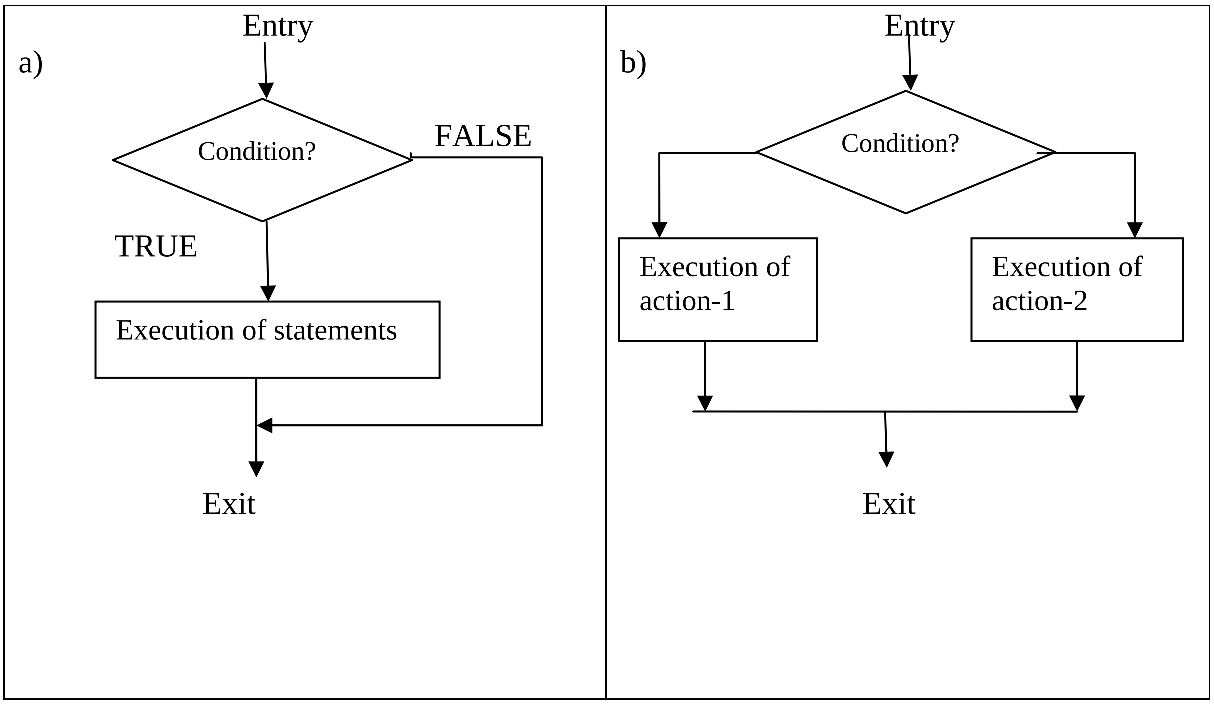


- **Decision or Test symbol:** (◇ , diamond symbol)

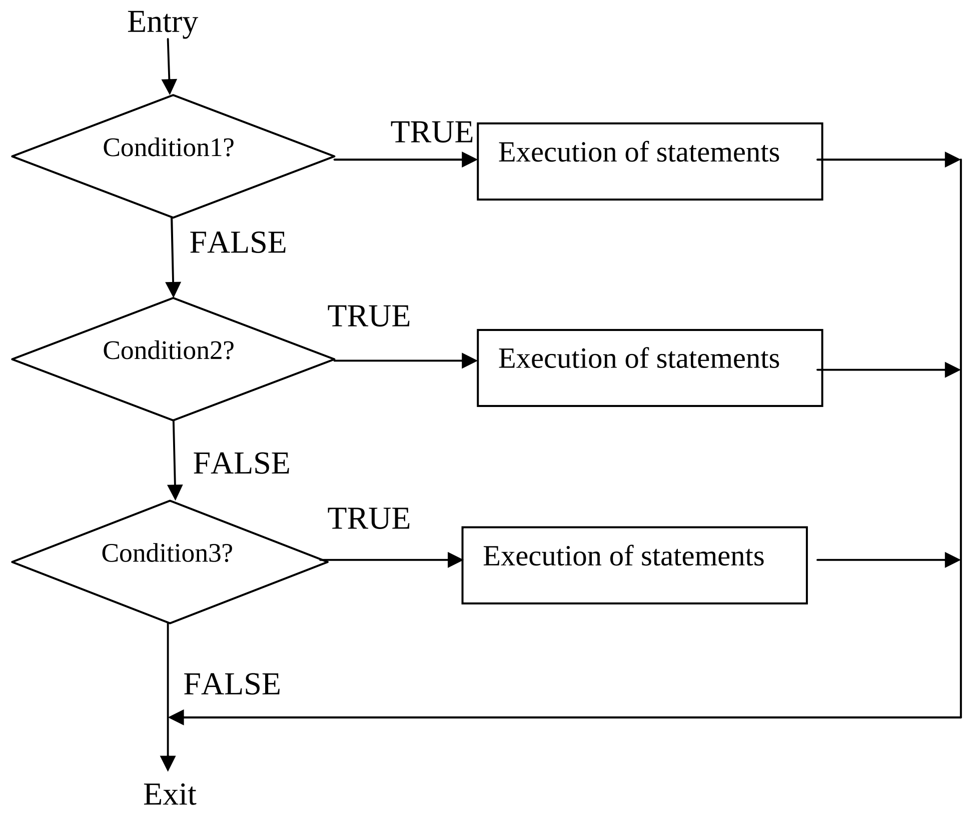
This symbol is used to take one of the decisions, depending on the condition. While solving a problem, one can take a single alternative or two or multiple alternatives depending on the situation.

a. single alternative decision: here more than one flow lines can be used depending upon the condition. It is usually in the form of a 'yes' or 'no' questions, with branching flow lines depending upon the answer.

b. Two alternative decisions: Two alternative paths are shown. On satisfying the condition statement(s) pertaining to action-1 will be executed, otherwise the other statement for action-2 will be executed.

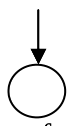


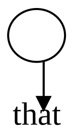
c. <u>Multiple alternative decisions:</u> Every decision block has two branches. In case the condition is satisfied execution of statements of appropriate block takes place, otherwise next condition will be verified. If condition 1 is satisfied then block1 statements are executed. In the same way other decision blocks are executed.



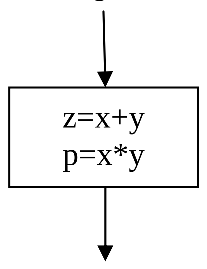
- **<u>Connector symbol :</u>** (○ , circle symbol)

It is used to establish the connection, whenever it is impossible to directly join two parts in a flow charts. When flow chart is on two separate pages then connector symbol is used for joining two parts. Only one flow line is shown with this symbol. Only connector names are written inside the symbol, that is , alphabets or numbers.


Connector for connecting to the next block

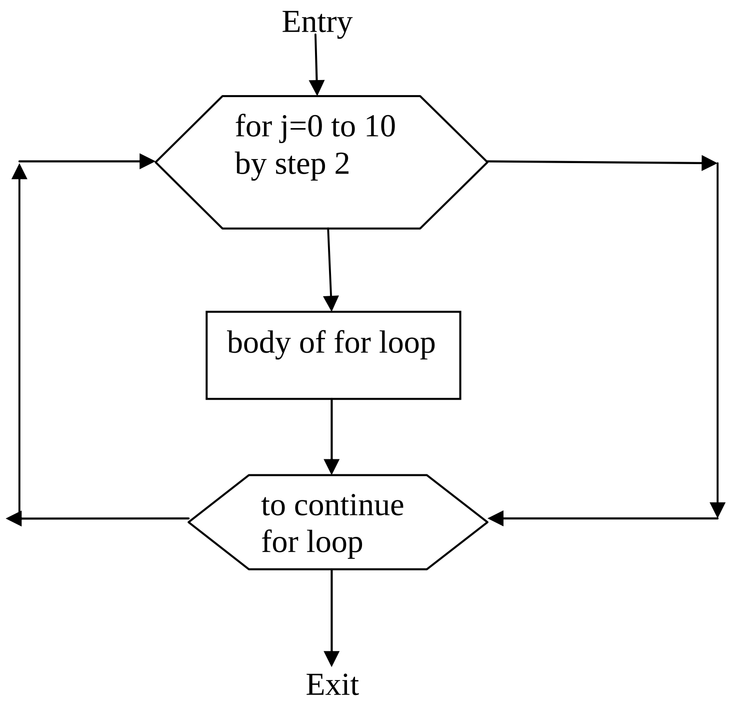

connector that comes from the previous block

- **<u>Process symbol:</u>** The symbol of process block should be shown a rectangle. It is usually used for data handling and values are assigned to the variables in this symbol.

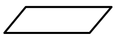


- **Loop symbol:** (,hexagon)

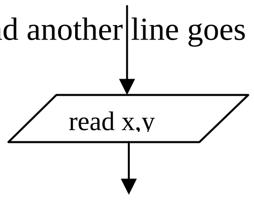
  Four flow lines are associated with this symbol, Two lines are used to indicate the sequence of the program and the remaining two are used to show the looping area, thatis, from the beginning to the end.



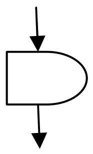
In above example, the for loop is repeated until j is incremented to 10.

- **Input/Output symbol:** ( ,parallelogram)

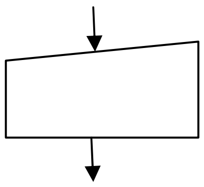
  Used to input/output the data; while the data is provided to the program for processing this symbol is used. There are two flow lines connected with the input/output symbol. One line comes to this symbol and another line goes from this symbol.



- **Delay symbol:** Symbol of delay is just like 'AND' gate. It is used for adding delay to the process. It is associated with two lines. One is incoming and another is outgoing.



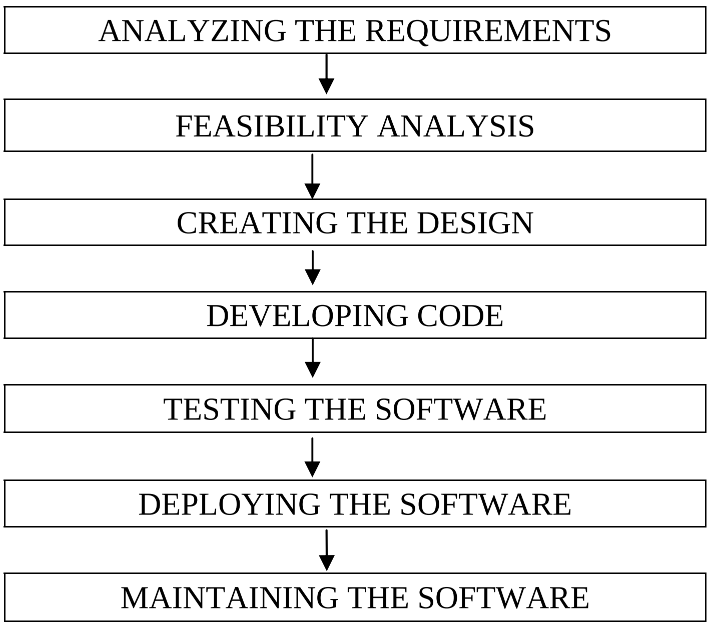
- **Manual input symbol:** This is used for assigning the variable values through the keyboard, whereas in the data symbol the values are assigned directly without manual intervention.



THE SOFTWARE DEVELOPMENT METHOD:

Programming is a problem solving activity. Programmers use the following software development methods sequence for developing efficient Software:

- Specifying the problem requirement (**requirements**)
- Analyze the problem (**analysis**)
- Designing an algorithm to solve the problem (**design)**
- Implement the algorithm (**Coding**)
- Test and verify the completed program (**Testing and validating**)
- Documentation and maintenance



1. ***Specify the problem requirements:*** forces you to state the problem clearly and unambiguously and to gain a clear understanding of what is required for its solution. Find more information from the person who posed the problem. If the requirements are not properly understood, then the software is bound to fall short of end user’s expectation. There should be continuous interaction between the software development team and end users. The task of requirement analysis is typically performed by a business analyst.
2. ***Analyze the problem:*** analyzing the problem involves identifying the problem.

   a. inputs, that is the data you have to work with

   b. outputs, the desired results

   c. any additional requirements or constraints on the solution.

   At this stage, we should also determine the required format in which the results should be displayed and develop a list of problem variables and their relationships. These relationships may be expressed as formulas.

   The process of modeling a problem by extracting the essential variables and their relationships is called <u>abstraction.</u>

3. ***Design the algorithm to solve the problem*:**

   Requires developing a list of steps called a algorithm to solve the problem and to then verify that the algorithm solves the problem as intended. Writing the algorithm is often the most difficult part of the problem solving process.

   Use top-down approach for writing the algorithm. In top-down design (also called divide and conquer) first list the major steps, or sub-problems, that need to be solved. Then solve the problem by solving its sub-problems.

   Most computer algorithms consists of at least the following sub-problems

   1. Get the data 2. Perform the computation 3. Display the results

   Algorithm refinement: development of a detailed list of steps to solve a particular step in the original algorithm.

   Desk checking: the step-by-step simulation of the computer execution of an algorithm.

4. ***Implementing the algorithm:***

   At this stage, one can begin to code the detailed program designs into program instructions of a given language. If all the previous steps have been completed properly, this coding should be almost ‘automatic’. The chances are high that a fairly successful program will result first time around. Although it may still contain bugs, these should be fewer and relatively easy to identify and correct.

5. ***Test and verify the completed program:***

   Testing and verifying the program requires testing the completed program to verify that it works as desired. Don’t rely on just one test case. Run the program several times using different sets of data to make sure that it works correctly for every situation provided for in the algorithm.

   There are two important activities that are performed while testing, they are verification and validation. Verification is a process of checking software based on some pre-defined specifications. Validation is testing the product to ascertain whether it meets user requirements.

6. ***Maintain and update the program:***

   Maintaining and updating the programs involves modifying a program to remove previously undetected errors and to keep it up to date as government regulations or company policies change.

   A disciplined approach is essential to create programs that are easy to read, understand and maintain. This includes <u>Documentation.</u> Documentation is the process of collecting, organizing and maintaining, in written the complete information of the program for future references.

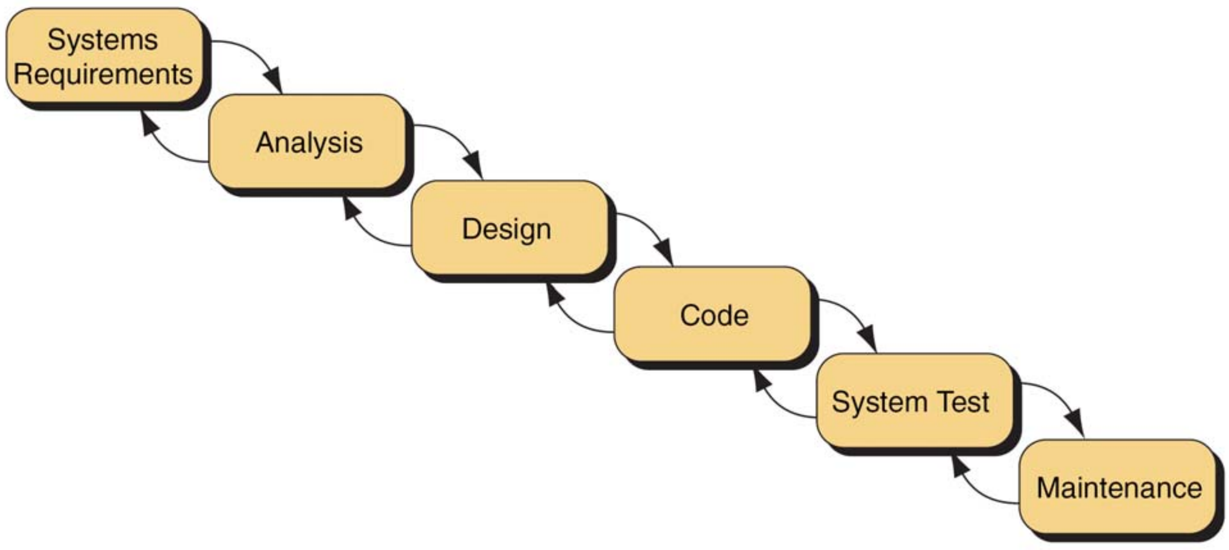


Fig: WATER FALL MODEL OF SOFTWEARE DEVELOPMENT

SOFTWARE ENGINEERING: is the establishment and use of sound engineering methods and principles to obtain software that is reliable and that works on real machines.

MODES OF OPERATION

There are two different ways that a large computer can be shared by many different users. These are the ***batch mode*** and the ***interactive mode.*** Each has its own advantages for certain types of problems.

Batch Processing

In ***batch processing,*** a number of jobs are entered into the computer, stored internally, and then processed sequentially. (A ***job*** refers to a computer program and its associated sets of input data.) After the job is processed, the output, along with a listing of the computer program, is printed on multiple sheets of paper by a high-speed printer. Typically, the user will pick up the printed output at some convenient time, after the job has been processed.

In *classical batch processing* (which is now obsolete), the program and the data were recorded on ***punched cards.*** This information was read into the computer by means of a mechanical card reader and then processed. In the early days of computing, all jobs were processed in this manner.

Modern batch processing is generally tied into a timesharing system. Thus, the program and the data are typed into the computer via a ***timesharing terminal*** or a personal computer acting as a terminal.

The information is then stored within the computer’s memory and processed in its proper sequence. This form of batch processing is preferable to classical batch processing, since it eliminates the need for punched cards and allows the input information (program and data) to be edited while it is being entered.

Large quantities of information (both programs and data) can be transmitted into and out of the computer very quickly in batch processing. Furthermore, the user need not be present while the job is being processed.

Therefore, this mode of operation is well-suited to jobs that require large amounts of computer time or are physically lengthy. On the other hand, the total time required for a job to be processed in this manner may vary from several minutes to several hours, even though the job may require only a second or two of actual computer time. (Each job must wait its turn before it can be read, processed, and the results displayed.) Thus, batch processing is undesirable when processing small, simple jobs that must be returned as quickly as possible (as, for example, when learning computer programming).

<u>Timesharing</u>

Timesharing allows many different users to use a single computer simultaneously. The host computer may be a mainframe, a minicomputer or a large desktop computer. The various users communicate with the computer through their own individual terminals. In a modern timesharing network, personal computers are often used as timesharing terminals.

An individual timesharing terminal may be wired directly to the host computer, or it may be connected to the computer over telephone lines, a microwave circuit, or even an earth satellite. Systems in which personal computers are connected to large mainframes over telephone lines are particularly common. Such systems make use of ***modems*** (i.e., modulator/dernodulator devices) to convert the digitized computer signals into analog telephone signals and vice versa.

Timesharing is best suited for processing relatively simple jobs that do not require extensive data transmission or large amounts of computer time. Many applications that arise in schools and commercial offices have these characteristics. Such applications can be processed quickly, easily, and at minimum expense using timesharing.

Interactive Computing

Interactive computing is a type of computing environment that originated with commercial timesharing systems and has been refined by the widespread use of personal computers. In an interactive computing environment, the user and the computer interact with each other during the computational session. Thus, the user may periodically be asked to provide certain information that will determine what subsequent actions are to be taken by the computer and vice versa.

Programs designed for interactive computing environments are sometimes said to be *conversational* in nature. Computerized games are excellent examples of such interactive applications: This includes fast-action, graphical arcade games, even though the user’s responses may be reflexive rather than numeric or verbal.

TYPES OF PROGRAMMING LANGUAGES

There are many different languages can be used to program a computer. The most basic of these is ***machine language--****a* collection of very detailed, cryptic instructions that control the computer's internal circuitry. Very few computer programs are actually written in machine language. The disadvantage is that machine language is very difficult to work with and every different type of computer has its own unique instruction set. Thus, a machine-language program written for one type of computer cannot be run on another type of computer without significant alterations.

Usually, a computer program will be written in some ***high-level*** language, whose instruction set is more compatible with human languages and human thought processes. Most of these are ***general-purpose*** languages such as C. There are also various ***special-purpose*** languages that are specifically designed for some particular type of application. Some common examples are CSMP and SIMAN, which are special-purpose ***simulation*** languages, and LISP, a Zistprocessing language that is widely used for artificial intelligence applications.

As a rule, a single instruction in a high-level language will be equivalent to several instructions in machine language. This greatly simplifies the task of writing complete, correct programs. Furthermore, the rules for programming in a particular high-level language are much the same for all computers, so that a program written for one computer can generally be run on many different computers with little or no alteration. Thus, we see that a high-level language offers three significant advantages over machine language: ***simplicity, uniformity*** and ***portability*** (i.e., machine independence).

A program that is written in a high-level language must, however, be translated into machine language before it can be executed. This is known as ***compilation*** or ***interpretation,*** depending on how it is carried out. (Compilers translate the entire program into machine language before executing any of the instructions. Interpreters, on the other hand, proceed through a program by translating and then executing single instructions or small groups of instructions.) In either case, the translation is carried out automatically within the computer. Most implementations of C operate as compilers.

A compiler or interpreter is itself a computer program. It accepts a program written in a high-level language (e.g., C) as input, and generates a corresponding machine-language program as output. The original high-level program is called the ***source*** program, and the resulting machine-language program is called the ***object*** program. Every computer must have its own compiler or interpreter for a particular high-level language.

It is generally more convenient to develop a new program using an interpreter rather than a compiler. Once an error-free program has been developed, however, a compiled version will normally execute much faster than an interpreted version

DESIRABLE PROGRAM CHARACTERISTICS

A program has the following characteristics. These characteristics apply to programs that are written in ***any*** programming language, not just C.

1. ***Integrity.*** This refers to the accuracy of the calculations. It should be clear that all other program enhancements will be meaningless if the calculations are not carried out correctly. Thus, the integrity of the calculations is an absolute necessity in any computer program.
2. ***Clarity*** refers to the overall readability of the program, with particular emphasis on its underlying logic. If a program is clearly written, it should be possible for another programmer to follow the program logic without undue effort. It should also be possible for the original author to follow his or her own program after being away from the program for an extended period of time. One of the objectives in the design of C is the development of clear, readable programs through an orderly and disciplined approach to programming.
3. ***Simplicity***. The clarity and accuracy of a program are usually enhanced by keeping things as simple as possible, consistent with the overall program objectives. In fact, it may be desirable to sacrifice a certain amount of computational efficiency in order to maintain a relatively simple, straightforward program structure.
4. ***Efficiency*** is concerned with execution speed and efficient memory utilization. These are generally important goals, though they should not be obtained at the expense of clarity or simplicity. Many complex programs require a tradeoff between these characteristics. In such situations, experience and common sense are key factors.
5. ***Modularity.*** Many programs can be broken down into a series of identifiable subtasks. It is good programming practice to implement each of these subtasks as a separate program module. In C, such modules are written as functions. The use of a modular programming structure enhances the accuracy and clarity of a program, and it facilitates future program alterations.
6. ***Generality.*** Usually we will want a program to be as general as possible, within reasonable limits. For example, we may design a program to read in the values of certain key parameters rather than placing fixed values into the program. As a rule, a considerable amount of generality can be obtained with very little additional programming effort.

STRUCTURED PROGRAMMING

The term *block-structured* applied to a computer language. Although the term block-structured language does not strictly apply to C, C is commonly referred to simply as a ***structured* language**. It has many similarities to other structured languages, such as ALGOL, Pascal, and Modula-2.

NOTE

The reason that C is not, technically, a block-structured language is that block structured languages permit procedures or functions to be declared inside other procedures or functions. However, since C does not allow the creation of functions within functions, it cannot formally be called block-structured

Here are some examples of structured and non-structured languages:

Non-structured	Structured
FORTRAN BASIC COBOL	Pascal Ada C++ C Java

There are three main principles of structured programming:

1. Program design using top-down or bottom up approach
2. Decomposition of program into components i.e. modular programming
3. Structuring of control flow

PROGRAM DESIGN:

- ✓ The program design concentrates on planning the solution as a collection of sub-solutions. It can be done in two ways- top down or bottom up design.
- ✓ During the top down design the divide and conquer policy is followed. The problem is divided into smaller sub-problems. The sub-problems are further divided into even smaller sub-problems.
- ✓ In top down approach, the calling component is always designed before its sub-components.
- ✓ The bottom up design is reverse of top down approach. Here, the process starts with the identification of the smallest sub-components of the total program. Such smallest components are combined to reach to a more abstract level and plan components of the higher level.
- ✓ The main drawback of this approach is that it is rarely possible to identify smallest sub-components needed for the program, especially for bigger programs.

MODULAR PROGRAMMING:

- ✓ The distinguishing feature of a structured language is *compartmentalization* of code and data. This is the ability of a language to section off and hide from the rest of the program all information and instructions necessary to perform a specific task.
- ✓ The modular programming principle suggests writing the code as a collection of many modules. A module is a portion of the program which is responsible to carry out a certain work and can be used with other modules.
- ✓ A module can be repeatedly called with different input values and can also be reused.
- ✓ All the modules are developed separately and the superior module can call the subordinate modules. The superior module is called the ***calling module*** and the sub ordinate module is called the ***called module***.

STRUCTURING OF CONTROL FLOW:

- ✓ In Structured programming there is a systematic flow of control throughout the program.
- ✓ It strongly advocates the principle of single entry and single exit to all sub-parts of the program.
- ✓ The advantage is that once the control enters the sub-part, it executes the statements within the sub-part until it reaches the exit point. This can be achieved by implementing all processing requirements using combination of three basic control flows:
  1. Sequence (steps one after the other)
  2. Selection (choose one out of two/many)
  3. Iteration (repeat steps many times)
- ✓ A structured language offers a variety of programming possibilities. For example, structured languages typically support several loop constructs, such as **while**, **do-while**, and **for**. In a structured language, the use of **goto** is either prohibited or discouraged and is not the common form of program control

INTRODUCTION TO C:

C is a general-purpose, structured programming language developed at A.T& T's Bell Laboratories of USA in 1972 by **Dennis Ritchie**. It was designed as a language in which to write the UNIX Operating system. It was originally use primarily for systems programming. Over the years, however, the power and flexibility of C, together with the availability of high-quality C compilers for computers of all sizes, have made it a popular language in industry for a wide variety of applications. C has the features of both BASIC and PASCAL. As a middle level language, C allows the manipulation of bits, bytes and addresses of the basic elements with which computer functions.

History & Evolution of 'C' language:

- Ken Thompson created a language which was based upon a language known as BCPL and it was called as B.
- B language was created in 1970, basically for UNIX operating system; Dennis Ritchie used ALGOL, BCPL and B as the basic reference language from which he created C.
- C was largely confined to use within Bell Laboratories until **1978,** when Brian Kernighan and Ritchie published a definitive description of the language. The Kernighan and Ritchie description is commonly referred to as "K&R C."
- By the mid **1980s,** the popularity of C had become widespread.
- In 1983, the American National Standards Institute (ANSI) began the definition of a standard of a C. It was approved in December 1989 which is known as ANSI C. In 1990, The International standards Organization (ISO) adopted the ANSI standard. This version of C is known as C99.

Language	**year**	**Proposed by**
ALGOL	1960	International Group
BCPL	1967	Martin Richards
B	1970	Ken Thompson
TRADITIONAL C	1972	Dennis Ritchie
K&R C	1978	Kernighan and Ritchie
ANSI C	1989	ANSI Committee
ANSI/ISO C	1990	ISO Committee
C99	1999	Standardization Committee

The C language is often described as a middle level language, because it combines the best features of high level languages with the control and flexibility of assembly language.

Important features of C language:

1) It is a robust language, whose rich set of built-in functions and operators can be used to write any complex program.

2) Programs written in C are efficient and fast. This is due to its variety of data types and powerful operators.

3) 'C's code is very portable, because 'C' is a machine-independent language, in the sense that it is easy to adapt software written for one type of computer or operating system to another type.

4) 'C' has very small key words (only 32). Its strength lies in its built-in functions. These built-in functions can be used for developing programs.

5) 'C' language is well suited for structured programming, thus requiring the user to think of a problem in terms of functions (or) blocks. A proper collection of these functions would make a complete program. This modular structure makes program debugging, testing and maintenance easier.

6) Another important feature of C is its ability to extend itself, i.e.. We can make our own functions and add these to C library.

7) C language is case sensitive, i.e. it differentiates the uppercase and lowercase letters. Ex: the identifier 'X' is different to 'x'.

8) 'C' language allows reference to a memory location with the help of pointer which holds the address of the memory location.

9) 'C' language allows dynamic allocation of memory i.e. a program can request the operating system to allocate/release memory.

10) 'C' language allows manipulations of data at the lowest level i.e. bit level manipulation. This feature is extensively useful in writing system software programs.

Basically a C program is a collection of functions that are supported by the C library. We can add our own functions to the C library. With the availability of a large number of functions, the programming task becomes simple.

Note: Function is nothing but a set of instructions, used to do some task.

BASIC STRUCTURE OF C PROGRAM:

A ‘C’ program can be viewed as a group of building blocks called functions. A function is a sub-routine that may include one or more statements designed to perform a specific task. To write a ‘C’ program we first create functions and then put them together. A ‘C’ program may contain a one or more sections as given below.

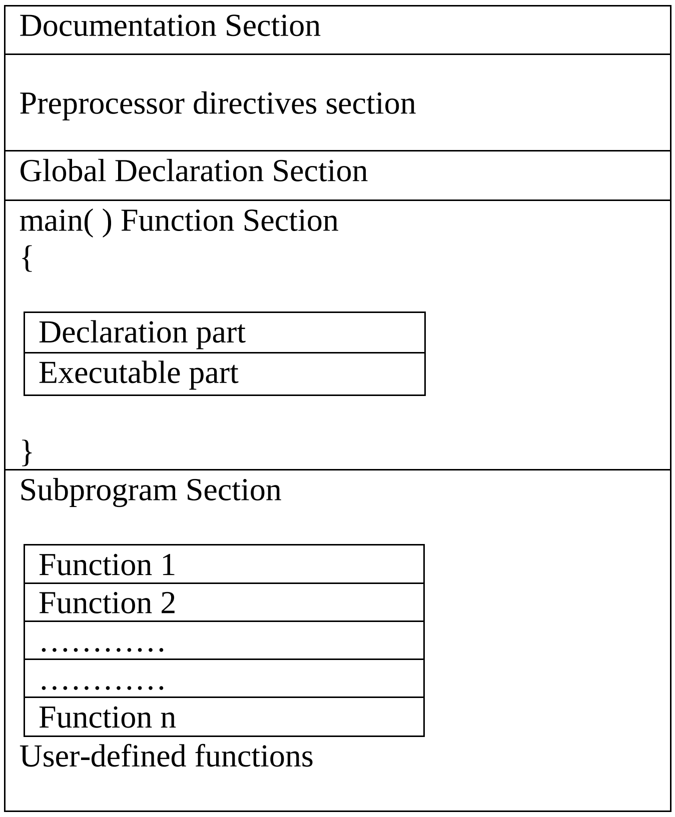


Fig: Basic Structure of a C program

1. **The Documentation Section:** It 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.

2. **Preprocessor Directives:** The preprocessor directives are the commands that give instructions to the C preprocessor, whose job it is to modify the text of a C program before it is compiled. A preprocessor directive begins with a number symbol (#) as its first non blank character. The most common directives are **#include and #define.**

   The #include directive instructs the C compiler to add the contents of an include file into your program during compilation. An *include file* is a separate disk file that contains information needed by your program or the compiler. Several of these files (sometimes called *header files*) are supplied with your compiler. Include files should all have an .H extension (for example, STDIO.H).

Header file:
- It is a pre-defined program.
- It contains functions (Sub programs), variables and constants etc.

Syntax: *# include <header file>*
Here, the symbol '#' reps. pre-processor. The word "include" is a system code.
Eg. #include <stdio.h>   Here, *stdio.h* is the name of header file.

Constant macro: a name that is replaced by a particular constant value before the program is sent to the compiler.

- Generally it can be placed at the beginning of the program. *It can be used in body of the program also*. This statement is called a 'Macro definition' or simply a 'Macro'.

Syntax: #define template expansion

Here, # is pre-processor symbol; define is system code

Eg. #define PI 3.143

#define P printf

#define M main( ) etc.

Note: Generally Templates can be used as Uppercase letters. Easy to identify the templates in a program.

#define pi 3.143 this directive instructs the pre processor to replace each occurrence of pi in the text of the C program by 3.143 before compilation begins

Most C programs require one or more include files. When using more than one directive each must appear on a separate line.

3. **Global Declaration Section:** 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 functions. This section also declares all the user-defined functions.

4. **main ( ) Function Section:** The execution of every C program starts from the function called main and must have only one main ( ) function.

   a) The ‘main ( )’ is a special function used by the C system to tell the computer where the program starts.

   b) Every program must have exactly one main function

   c) Opening brace ‘{‘and closing brace ‘}’ are the delimiters of any function

   d) All the statements between these two braces are called as function body. Function body has two parts: **declaration and executable statements**. The declarations tell the compiler what memory cells are needed in the function. The executable statements are translated into machine language and later executed.

   e) The default data type of main function is int .It means main returns integer value to the operating system. The sub program 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.

Note: All sections, except the main function section may be absent when they are not required.

COMMENT: Any part of your program that starts with /* and ends with */ is called a *comment*. The compiler ignores all comments, so they have absolutely no effect on how a program works. You can put anything you want into a comment, and it won't modify the way your program operates. There are two types of comment line: a) single line comment b) Multi line comment. A comment can span part of a line, an entire line, or multiple lines. Here are three examples:

a) Single line comment is used to represent a single statement as a comment.

```
Ex:     /* A single-line comment */
        int a,b,c; /* A partial-line comment */
```

b) Multi line comments are used to represent single or multi statements as a comments.

```
Ex:     /* a comment
        spanning
        multiple lines */
```

However, you shouldn't use *nested* comments (in other words, you shouldn't put one comment within another).

Most compilers would not accept the following:

```
/*
/* Nested comment */
*/
```

Many beginning programmers view program comments as unnecessary and a waste of time. This is a mistake! The operation of your program might be quite clear while you're writing it--particularly when you're writing simple programs. However, as your programs become larger and more complex, or when you need to modify a program you wrote six months ago, you'll find comments invaluable. Now is the time to develop the habit of using comments liberally to document all your programming structures and operations.

Simple ‘C’ Programs:

Ex 1: C program to print a message.

```
/*Simple C Program */
#include<stdio.h>
#include<conio.h>
void main ( )
{
    /*prints the string */
    printf(“HELLO KNREDDY”);
 }
```

```
/ * program to calculate the area of a c i r c l e */
#include <stdio.h>
void main(  )
{
            float radius, area;
            printf("Radius =?");
            scanf ("%f",&radius) ;
            area = 3.14159 * radius * radius;
            printf("Area = %f" ,area);
}
```

The following features should be pointed out in the above program.

1. The program is typed in lowercase. Either upper- or lowercase can be used, though it is customary to type ordinary instructions in lowercase.

2. The first line is a comment that identifies the purpose of the program.

3. The second line contains a reference to a special file (called **stdio .h)** which contains information that must be included in the program when it is compiled.

4. The third line is a heading for the function **main.** The empty parentheses following the name of the function indicate that this function does not include any arguments.

5. The remaining five lines of the program are indented and enclosed within a pair of braces. These five lines comprise the compound statement within **main.**

6. The first indented line is a ***variable declaration.*** It establishes the symbolic names **radius** and **area as *floating-point variables*** (more about this in the next chapter).

7. The remaining four indented lines are expression statements. The second indented line **(printf)** generates a request for information (namely, a value for the radius). This value is entered into the computer via the third indented line **(scanf).**

8. The fourth indented line is a particular type of expression statement called an ***assignment statement.*** This statement causes the area to be calculated from the given value of the radius. Within this statement the asterisks (*) represent multiplication signs.

9. The last indented line **(printf)** causes the calculated value for the area to be displayed. The numerical value will be preceded by a brief label.

10. Notice that each expression statement within the compound statement ends with a semicolon. This is required of all expression statements.

11. Finally, notice the liberal use of spacing and indentation, creating ***whitespace*** within the program. These features are not grammatically essential, but their presence is strongly encouraged **as** a matter of good programming practice.

ENTERING THE PROGRAM INTO THE COMPUTER:

- ✓ The program is to be entered into computer for compilation and running.
- ✓ In older versions of C this was done by typing the program into a text file on a line by line basis using a text editor or a word processor.
- ✓ Modern C or C++ includes a screen editor, usually integrated into the software environment.
- ✓ The manner in which to access the editor varies from one version to other.
- ✓ For example, for TURBO C++ click on the TURBO C++ icon or enter by using DOS commands.
- ✓ This will result a window with a title bar and menu bar; and editing area beneath the menu bar for editing.
- ✓ Selecting one of the items in the menu bar will cause a drop-down menu to appear, with a number of choices related to the menu bar selection.
- ✓ Usually pointing device, such as a mouse is used to select a menu item.
- ✓ Scroll bars are present beneath and to right of the editing area. The scroll bars allow you to move quickly to other parts of the program if the program listing extends beyond the confines of the screen.
- ✓ Finally, the last line is the status bar, which indicates the current status of the editing area, or purpose of the currently highlighted menu selection.
- ✓ Once the program has been entered it should be saved before it is executed (with .C extension). Once the program has been saved and a name has been provided it; can be again saved at some time later (for saving recent editing).

COMPILING AND EXECUTING THE PROGRAM:

- Once the program has been entered into the computer, edited and saved, it can be compiled and executed by selecting RUN from the debug menu.
- If the program does not compile successfully, a list of error messages will appear in a separate window. Each error message indicates the line number where the error was detected as well as the type of error.
- If the program does compile successfully, however it will immediately begin to execute prompting for input, displaying output etc., within the new window.

C TOKENS:

The smallest element identified by the compiler in a source file is called a TOKEN. It may be a single character or sequence of characters to form a single item. Tokens can be classified as

a) Keywords (ex: float, while etc...)
b) Identifiers (ex: main, amount etc…),
c) Literals (or constants) (ex:-15.5, 100 etc…),
d) Operators (ex: + - *, etc…),
e) Strings (ex: “ABC”,”year”,”A” etc…),
f) Special symbols ([ ] { } etc…)

Key words: Keywords are the words whose meaning has already been explained to the C compiler (or in a broad sense to the computer). The keywords cannot be used as variable names because if we do so we are trying to assign a new meaning to the keyword, which is not allowed by the computer. The keywords are also called ‘Reserved words’.

There are only 32 keywords available in C.

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

IDENTIFIERS: come in two varieties: standard identifiers and user defined identifiers

STANDARD IDENTIFIER: a word having special meaning but one that a programmer may redefine (but redefinitions are not recommended). eg: printf and scanf

USER-DEFINED IDENTIFIERS: Identifiers allow us to name data and other objects in the program. Each identified object in the computer is stored at a unique address.

Identifiers refer to the names of variables, functions and arrays.

The identifiers must conform to the following rules:

1. First character must be an alphabet (or underscore)
2. Identifier names must consist of only letters, digits and underscore.
3. Any standard C language keyword cannot be used as a variable name.
4. An identifier defined in a C standard library should not be redefined.
5. No commas or blanks are allowed within a variable name.
6. No special symbol other than an underscore (as in gross_sal) can be used in a variable name.
7. Case matters (that is, upper- and lowercase letters). Thus, the names count and Count refer to two different variables

Some examples of identifiers and their correctness given in table:

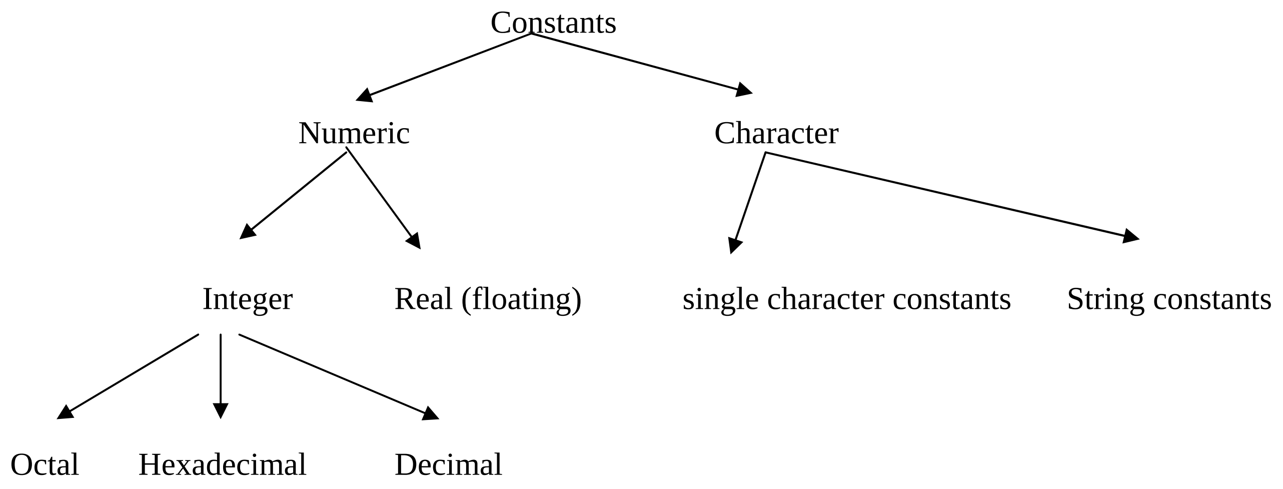
IDENTIFIERS	**LEGALITY**
Percent	Legal
y2x5__fg7h	Legal
annual_profit	Legal
_1990_tax	Legal but not advised
savings#account	Illegal: Contains the illegal character #
2names	Illegal: first character is digit
sum-salary	Illegal: contains hyphen
std number	Illegal: contains space
double	Illegal: Is a C keyword
$sum	$ is not allowed(special symbols)

Because C is case-sensitive, the names percent, PERCENT, and Percent would be considered three different variables

A variable may take different values at different times during the execution.

ANSI standard recognizes a length of 31 characters. However, length should not be more than eight characters, since only the first eight characters are treated as significant by many compilers.

CONSTANTS: The alphabets, numbers and special symbols are properly combines to form constants and variables. Constants are fixed values that do not change during the execution of program. Several types of constants are:



For example in the equations 5x+2y=45 since 5, 2 and 45 cannot change , these are called constants, where as the quantity X&Y can vary or change hence these are called variables .

Numeric constants:

i) Integer constants: It refers to a sequence of digits, it has to follow the below rules:

1. Integer constants must have at least one digit
2. It must not have a decimal point
3. It could be either positive or negative
4. If no sign precedes an integer constant it is assumed to be positive
5. No commas, blank space are allowed.
6. The allowable range for integer constants is -32768 to +32767 (16-bit machine) integer constants can be specified in decimal, octal, or hexa decimal notation.

a) A decimal integer constant: It consists of sequence of one or more decimal digit 0 through 9 preceded by an optional – (or) + sign. The first digit of the sequence cannot be 0 unless the decimal integer constant is 0.

Ex: 0 276 3412 31467 -7123

Note: Embedded spaces, commas, and non-digit characters are not permitted between digits.

Ex: 12 727 23,879 $1772 are illegal numbers.

b) An Octal Integer constant: It consists of any combination of digits from the set 0 through 7, with a leading 0.

Ex: 012 07134 07777

c) A hexa Decimal integer constants: It consists of the digit 0, followed by one of the letter x (or) X, followed by a sequence of one more hexadecimal digits 0 through 9 or letter a through f (or) A through F represent the numbers 10 through 15.

Ex: 0X1F 0XABC 0X9a2F 0XFFFF

Note: we rarely use octal and hexa decimal numbers in programming.

ii) Real Constant (floating point constant): A real constant are sequence of digits with a decimal point (fractional part) like 45.382.Such numbers are called real (or floating point) constants.

<u>Rules for constructing Real Constants</u>:

1. A Real constant must have least one digit.
2. It must have a decimal point.
3. It could be either positive or negative.
4. Default sign is positive.
5. No commons, black space are not allowed.

Ex: 1.01 0.712 34.576 -7.123

These numbers are shown in *decimal notation*, having a whole number followed by a decimal point. It is possible to omit digits before the decimal point or digits after the decimal point.

Ex: 215.0 0.39 -0.92 +5.0 are valid real numbers.

The real numbers may also be expressed in exponential (or, scientific) notation.

For example, the value 215.65 may be written as 2.1565e2 in exponential notation. (E2 means multiply by 10

The general form:

$\boxed{\text{mantissa } \mathbf{e}^{\text{exponent}}}$

- The *mantissa* is either a real number expressed in decimal notation or an integer.
- The *exponent* is an integer number with an optional + or – sign.
- The letter *e* separating the mantissa and the exponent can be written in either lowercase or uppercase

The scientific notation is often used to express numbers that are either very small or very large.

Ex: 7500000000 may be written as 7.5e9 or 75e8.

Similarly,-0.000000368 is equivalent to -3.68E-7.

(Coefficient) e $^{(integer)}$ = (coefficient) * 10$^{(integer)}$

iii) Character constants:

a) Single character constants:

Rules for constructing character constants:

1. A character constant is a single alphabet, a single digit or a single special symbol enclosed with in a pair of single inverted commas. Both the inverted commas should point to the left. For example 'A' is not valid character constant where as 'A' is valid.
2. Character constants have integer values known as **ASCII** (AMERICAN STANDARD CODE FOR INFORMATION INTERCHANGE) values.
3. The valid range of a character constant -128 to127. It appears surprising that the character constant should have a numeric range. Character and integer constant are often used interchangeably. For example 'A' and 65 are one and the something, since when we say 'A' it is replaced by ASCII value, which is 65.

   Example; '0' 'A' 'F' 'Y'

THE C 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 special characters are listed below.

+	-	*	/	=	%	&	#
!	?	^	"	'	-	\	|
<	>	(	)	[	]	{	}
:	;	.	,	_	(blank space)		

Most versions of the language also allow certain other characters, such as @ and $, to be included within strings and comments.

C uses certain combinations of these characters, such as **\b, \n** and \t, to represent special conditions such **as** backspace, newline and horizontal tab, respectively. These character combinations are known as ***escape sequences.***

Backslash character constants:

C supports some special backslash character constants that are used in output functions. Each one of them represents one character, although they consist of two characters. These character combinations are known as *escape sequences.*

Constant	meaning
\a	Alert(bell)
\b	Backspace
\n	new line
\r	carriage return
\t	horizontal tab
'\\'	back slash
\"	double quotation
\'	single quote
\v	Vertical tab
\?	Question mark
\0	null

b) String constant: A string constant is a sequence of characters enclosed with in a pair of double inverted commas. Sometimes certain special characters (e.g., a backslash or a quotation mark) must be included as a part of a string constant. These characters *must* be represented in terms of their escape sequences. Similarly, certain nonprinting characters (e.g., tab, newline) can be included in a string constant if they are represented in terms of their corresponding escape sequences

Ex:"hello"   "1999"   "5+4+6"   "good bye\n"

DATA TYPES:

Each data type has predetermined memory requirement and an associated range of legal values. Every programming language has its own data types. Storage representations and machine instructions to handle constants differ from machine to machine.

ANSI C supports four classes of data types.

1. primary (or fundamental) data types
2. user defined data types
3. derived data types
4. Empty data set.

1. Primary data types:

All C compilers support four fundamental data types, namely integer (int), character (char), floating point (float), and double-precision point (double).

The interpretation of a qualified integer data type will vary from one C compiler to another, though there are some commonsense relationships

a) Integers*:*

C provides three different types of integers they are int, short int and long int. The basic data types can be augmented by the use of the data type ***qualifiers*** *short*, long, signed and unsigned. For example, integer quantities can be defined as short int, long int or unsigned int (these data types are usually written simply as short , long or unsigned, and are understood to be integers).

The difference between these three integers is the number of bytes. The variables of these types occupy and subsequently the range of values. A short int occupies 2 bytes, an int occupies 2 bytes and the long int occupies 4 bytes.

Type	**Size(Bytes required)**	**Range**
Short int	2	-32,768 to 32,767 (-2^{15} to $2^{15}-1$)
int	2	-32,768 to 32,767 (-2^{15} to $2^{15}-1$)
long int	4	-2,147,483,648 to 2,147,483,647
unsigned short int	2	0 to 65,535
unsigned int	2	0 to 65,535
unsigned long int	4	0 to 4,294,967,295

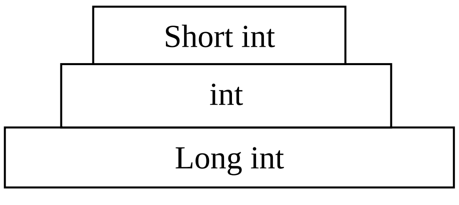


Fig: integer types

b) Float: Like integers floats are divided into three types. They are float, double and long double. The difference between these three floats are the number of bytes, the variable of these types occupy and subsequently the range of values. A float occupies 4 bytes, a double occupies 8 bytes and the long double occupies 10 bytes.

Type	**Description**	**Size(bytes)**	**Range**	**Precision**
Float	Single precession	4	3.4E-38 to 3.4E+38	6
Double	Double precession	8	1.7E-308 to 1.7E+308	14
Long double	Extended precession	10	3.4E-4932 to 3.4E+4932	More than 14

c) Characters: A char is a data type which can store an element of machine character set. A single character can be defined as a character (char) type data. Characters are usually stored in 8 bits (1 byte) of internal storage. The character set is usually the ASCII. These are two types; they are signed and unsigned characters. The differences between these two types are the range of values. Both will occupy one byte.

Type	Bytes required	Range
Signed char	1	-128 to 127
Unsigned char	1	0 to 255

2. User-defined data types: these are divided into two types

1) typedef
2) enumeration

i) typedef: The users can define an identifier that represent an existing data type by a feature known as "type definition". The user defined data type identifier can later be used to declare variables.

Syntax: typedef type identifier;

Where type refers to an existing data type and identifier refers to the new name given to the data type.

Ex: typedef int sno; typedef float salary;

Here sno symbolizes int and salary symbolizes float. These can be used to declare variables as follows.

sno c1,c2; salary e1,e2;

Note: The main advantage of typedef is that we can create meaningful data type names for increasing the readability of the program.

ii) Enumeration: Another user defined data type is enumerated data type provided by ANSI.

General form:

enum identifier {value 1, value 2, ……, value n};

The identifier is a user defined enumerated data type which can be used to declare variables that can have one of the values enclosed within the braces. After that we can declare variables to be of this new type.

enum identifier v1, v2, ….vn;

The enumerated variables v1, v2, …..vn can only have one of the value 1, value 2, ……value n.

Ex 1:

enum month {january, february, ….december};
enum month month_st, month_end;

(or)

enum month {january, february, …., December} month_st, month_end;

Here the declaration and definition of enumerated variables can be combined in one statement.

The compiler automatically assigns integer digits beginning with 0 to all the enumeration constants. That is, the enumeration constant Monday is assigned with 0, Tuesday is assigned with 1 and so on. However, the automatic assignments can be overridden by assigning values explicitly to the enumeration constants.

For example, enum day{Monday=1,Tuesday, ……., Saturday};

here, the constant Monday is assigned the value 1.The remaining constants are assigned values that increases successively by 1.

3) Derived data types: There are some derived data types which are supported by C such as arrays, functions, structures, and pointers. Derived data types will be explained later.

4) Empty data set: It is also known as void data types. It indicates that no other data types have been used with the given identifier.

VARIABLES AND ARRAYS:

A variable is a data name which can be used to store a data value and a variable may take different values at different times, during execution.

A *variable* is a named data storage location in your computer's memory. By using a variable's name in your program, you are, in effect, referring to the data stored there

The ***array*** is another kind of variable that is used extensively in C. An array is an identifier that refers to a ***collection*** of data items that all have the same name. The data items must all be of the same type (e.g., all integers, all characters, etc.). The individual data items are represented by their corresponding ***array-elements*** (i.e., the first data item is represented by the first array element, etc.). The individual array elements are distinguished from one another by the value that is assigned to a ***subscript.***

DECLARATION OF VARIABLES:

Variable declaration does two things.

i) It tells the compiler what the variable name is.

ii) It specifies what type of data the variable will hold.

Any variable used in the program must be declared before using it in any statement. The type declaration statement is usually written at the beginning of the C program.

Syntax: Data-type var1,var2 … var n;

Ex:
```
int i, count;
float price, salary;
char c;
```

STATEMENTS:

A ***statement*** causes the computer to carry out some action. There are three different classes of statements in C. They are ***expression statements, compound statements*** and ***control statements.***

An ***expression statement*** consists of an expression followed by a semicolon. The execution of an expression statement causes the expression to be evaluated.

Several expression statements are shown below.

```
a = 3;
c = a + b;
++i;
printf("Area = %f area) ;
;
```

The first two expression statements are assignment-type statements. Each causes the value of the expression on the right of the equal sign to be assigned to the variable on the left.

The third expression statement is **an** incrementing-type statement, which causes the value of **i** to increase by 1.

The fourth expression statement causes the **printf** function to be evaluated. This is a standard C library function that writes information out of the computer**.**

The last expression statement does nothing, since it consists of only a semicolon. It is simply a mechanism for providing **an** empty expression statement in places where this type of statement is required. Consequently, it is called a ***null statement.***

A ***compound statement*** consists of several individual statements enclosed within a pair of braces { }.

The individual statements may themselves be expression statements, compound statements or control statements. Thus, the compound statement provides a capability for embedding statements within other statements. Unlike an expression statement, a compound statement does ***not*** end with a semicolon.

A typical compound statement is shown below.

```
{
p i = 3.141593;
circumference = 2. * p i * radius;
area = p i * radius * radius;
}
```

Control statements are used to create special program features, such as logical tests, loops and branches.

Many control statements require that other statements be embedded within them, as illustrated in the following example.

The following control statement creates a conditional loop in which several actions are executed repeatedly, until some particular condition is satisfied.

```
while (count <= n)
{
        printf ( "x=" ) ;
        scanf ( "%f" , &x) ;
        sum += x;
        ++count;
}
```

SYMBOLIC CONSTANTS

A symbolic constant is a name that substitutes for a sequence of characters. The characters may represent a numeric constant, a character constant or a string constant. When a program is compiled, each occurrence of a symbolic constant is replaced by its corresponding character sequence.

Symbolic constants are usually defined at the beginning of a program.

A symbolic constant is defined by writing **#define** ***name*** ***text***

where ***name*** represents a symbolic name, and ***text*** represents the sequence of characters that is associated with the symbolic name. Note that ***text*** does not end with a semicolon, since a symbolic constant definition is not a true C statement. Moreover, if ***text*** were to end with a semicolon, this semicolon would be treated as though it were a part of the numeric constant, character constant or string constant that is substituted for the symbolic name.

A C program contains the following symbolic constant definitions.

#define TAXRATE 0.23

#define P I 3.141593

#define TRUE **1**

#define FALSE 0

#define FRIEND "KNREDDY"

Notice that the symbolic names are written in uppercase, to distinguish them from ordinary C identifiers. Also, note that the definitions do not end with semicolons.

Now suppose that the program contains the statement

area = PI * **radius** * **radius;**

During the compilation process, each occurrence of a symbolic constant will be replaced by its corresponding text. Thus, the above statement will become

area = 3.141593 * radius * radius;

Now suppose that a semicolon had been (incorrectly) included in the definition for PI, i.e.,

#define PI **3.141593;**

The assignment statement for **area** would then become

area = 3.141593; * radius * radius;

Note the semicolon preceding the first asterisk. This is clearly incorrect, and it will cause an error in the compilation.

The substitution of text for a symbolic constant will be carried out anywhere beyond the **#define** statement, ***except*** within a string. Thus, any text enclosed by (double) quotation marks will be unaffected by this substitution process.

The **printf** statement will be unaffected by the symbolic constant definition,

It is much easier to change the value of a single symbolic constant than to change every occurrence of some numerical constant that may appear in several places within the program.

The **#define** feature, which is used to define symbolic constants, is one of several features included in the C ***preprocessor*** (i.e., a program that provides the first step in the translation of a C program into machine language).

OPERATORS & EXPRESSIONS:

OPERATORS:

An operator is a symbol which represents a particular operation that can be performed on some data. The data itself is called the 'operand'. Expressions are made by combining operators between operand.

C operators are classified into below categories:

1. Arithmetic
2. Relational
3. Logical or Boolean
4. Assignment
5. Unary (Increment and decrement)
6. Bit wise
7. Conditional or ternary operator
8. Special operators

1. Arithmetic operators:

The arithmetic operators that we come across in 'C' language are +, -, *, /and %. All of these operators are called 'binary' operators as they operate on two operands at a time. Each operand can be an *int* or *float* or *char*.

Arithmetic operators

Operator	Meaning
+	Addition or unary plus
-	Subtraction or unary minus
*	Multiplication
/	Division
%	Modulo division.

Ex:

int x, y, z;   z=x+y;   z=x-y;   z=x*y;   z=x/y;

If both operands are integers, then the expression called an integer expression and the operation is called integer arithmetic. Integer arithmetic always yields an integer value.

Ex: for a=14,b=4,

a+b=18   a-b=10   a*b=56

a/b=3(decimal part truncated)

a % b=2(remainder of division)

During integer division, if the both the operands are of the same sign; the result is truncated towards to zero. If one of them is negative; the direction of truncation is implementation depended.
Ex: 6/7=0 and -6/-7=0 but -6/7=0 or 1(machine dependent).
Similarly in modular division, the sign of the result is always the sign of the first operand (the dividend),
Ex: -14%3= -2;   -14%-3= -2;   14%-3= 2;   14%3=2;

If both operands are real, then the expression is called a real expression and the operation is called real arithmetic. A real operand may be either in decimal or exponential notation. Real arithmetic always yields a real value. The modulus (%) operator cannot be used for real operands.

If one of the operand is real and other is integer then the expression is called mixed-mode arithmetic expression. Here only the real operation is performed and the result is always in real form.

```
/*  C Program to convert days into months*/
#include<stdio.h>
#include<conio.h>
void main( )
{
    int months,days;
    clrscr( );
    printf("Enter days \n");
    scanf("%d",&days);
    months=days/30;
    days=days%30;
    printf("months=%d\n days=%d",months,days);
    getch( );
}
```

Output:

```
Enter days 125
months=4
days=5
```

2. Relational operators:

The relational and equality operators are used to test or compare values between two operands. The relational and equality operators produce an integer result to express the condition of the comparison. If the condition is false then the integer result is 0.If the condition is true then the result is non-zero.

Relational operators

Operator	Meaning
<	is less than
<=	Is less than or equal to
>	Is greater than
>=	Is greater than or equal to
= =	Is equal to
!=	Is not equal to

Ex: 1) 10<20 result is TRUE (i.e.1)   2) 20<10 result is FLASE(i.e.0)
3) 10<=20 result is TRUE(i.e.1)   4) 20>=10 result is TRUE(i.e.1)
4) 10 = = 10 result is TRUE(i.e.1)   5) 10!=10 result is FALSE(i.e.0)

```
/*Program: Write a c program to find largest number among two numbers*/
#include<stdio.h>
#include<conio.h>
void main( )
{
        int a,b,max;
        clrscr( );
        printf("\n Enter any two numbers");
        scanf("%d%d",&a,&b);
        if(a>b)
                max=a;
        else
                max=b;
        printf(" %d is greater of %d,%d", max,a,b);
        getch( );
 }
```

3. Logical operators:

Logical operators are used to combine two or more relations. The logical operators are called Boolean operators. Because the tests between values are reduced to either true or false, with zero being false and one being true.

Operator	**Meaning**
&	Logical AND
| |	Logical OR
!	Logical NOT

(NOTE: Expression is nothing but, it is a combination of operators and operands (i.e, constants, variables etc.) which will give you some value as its result)

The expressions can be connected as: (expression 1) &&/|| (expression 2)

The below table demonstrates '&&' and '||' Operators:

Operands		Results	
Exp 1	Exp 2	Exp 1 && Exp 2	Exp 1 || Exp 2
0	0	0	0
0	Non zero	0	1
Non zero	0	0	1
Non zero	Non zero	1	1

From the above table following rules can be followed for logical operators:

- The logical AND (&&) operator provides true result when both expressions are true otherwise 0.
- The logical OR (||) operator provides true result when one of the expression is true otherwise 0.
- The logical NOT (!) operator provides 0 if the condition is true otherwise 1

/*program: Write a c program that illustrates the use of logical operators*/

```
#include<stdio.h>
#include<conio.h>
void main( )
{         clrscr( );
          Printf("\n condition: Return values\n");
          Printf("\n 5>3&&5<10  :%5d", 5>3&&5<10);
          Printf("\n 8>5 ||8<2  :%5d", 8>5 ||8<2);
          Printf("\n !(8==8)  :%5d",!(8==8));
          getch ( );
}
```

Output:

```
Condition    :     Return values
5>3&&5<10:     1
8>5 ||8<2      :   1
!(8==8)        :    0
```

/*Program: Write a c program to find largest number among three numbers*/
```
#include<stdio.h>
#include<conio.h>
void main( )
{
         int a,b,c,max;
         clrscr( );
         printf("\n Enter any three numbers");
         scanf("%d%d%d",&a,&b,&c);
        if(a>b&&a>c)
                max=a;
        else if(b>c)
               max=b;
        else
              max=c;
       printf("\n%d is maximum of %d,%d,%d ", max,a,b,c);
       getch( );
 }
```

4. Assignment Operators: it is used to assign the result of an expression to a variable.

Values can be assigned to variables using the assignment operator '=' as follows:

```
variable_name=constant;
```

Ex: balance=1278;
status='Y';

C permits multiple assignments in one line.

Ex: balance=1278; Yes='x'; are valid statements.

An assignment statement implies that the value of the variable on the left of the 'equal sign' is set equal to the value of the quantity (or the expression) on the right.

It is also possible to assign a value to a variable at the time the variable is declared. This takes the below form:

int x=10;

A floating point is truncated if a real value is assigned to an integer variable.

In addition C has a set of 'shorthand' assignment operators.

Syntax: V op= exp

Here V is a variable, exp is an expression and op is a binary arithmetic operator. The operator op = is known as the shorthand assignment operator. The following are the shorthand assignment operators.

+= add assignment operator

-= minus assignment operator

*= multiply assignment operator

/= divide assignment operator

%= modulus assignment operator

Ex: x+ = y is equivalent to x= x + y

x- = y is equivalent to x= x - y

x*=y is equivalent to x=x*y

x/=y is equivalent to x=x/y

x%=y is equivalent to x=x%y

```
/* Program to find sum of first ten numbers*/
#include<stdio.h>
#include<conio.h>
void main(0
{
        int i,sum;
        clrscr( );
        sum=0;
        for(i=1;i<=10;i++)
                sum+=I;
        printf("\n Sum of first 10 numbers= %d",sum);
}
```

5. Unary operators:

C includes a class of operator that act upon a single operand to produce a new value. Such operators are known as Unary operators. Unary operators usually preceded their single operand. Most commonly used unary operators are

1) Unary minus operator

2) Increment and Decrement operators.

a) Unary minus: Where a minus sign precedes numerical constants, variables or an expression. Whereas the unary minus operation is different from the arithmetic operator, which do Dot Subtraction. Thus a negative number is actually an expression consisting of unary minus operator.

Ex: x=10, y=-x; result is -10.

b) Increment and Decrement operators: The increment (++) and Decrement (--) operators are add one and subtract one. These are unary operators since they operate on only one operand. The operand has to be a variable. The increment or decrement of the value either before or after the value of the variable is used. If the operator appears before the variable, it is called a prefix operator. If the operator appears after the variable, it is called a postfix operator.

Operator	Meaning
a ++	Post increment
++a	Pre increment
a--	Post decrement
--a	Pre decrement

a++ and ++a is the same when the statements are independent like

```
a=5;            a=5;
a++;            ++a;
```

In the both cases a value will be 6.

When the prefix ++ (or--) is used in an expression, the variable is incremented (or decremented) first and then the expression is evaluated using with the new value of the variable. Whereas the postfix ++ (or --) is used in an expression, the expression is evaluated first using with the original values of the variables and then the variable is incremented (or decremented) by one.

Consider the following:

```
a=5;            b=a++;
```

In this case the value of a would be 6 and b would be 5.If we write the above statement as

```
a=5;            b=++a;
```

In this case the value of a would be 6 and b would be 6.

```
/* program to demonstrate post increment and decremenrt*/
#include<stdio.h>
#include<conio.h>
void main( )
{
        int a, b, x, y;
        clrscr( );
        a=4;b=5;x=10;y=20;
        printf("value of a :%d\n",a);
        printf("value of a++ :%d\n",a++);
        printf("new value of a :%d\n",a);
        printf("value of b :%d\n",b);
        printf("value of b-- :%d\n",b--);
        printf("value of b :%d\n",b);
        printf("value of x*y++ :%d\n",x*y++);
        getch( );
}
```

Output:

```
value of a :4
value of a++ :4
new value of a :5
value of b :5
value of b-- :5
value of b :4
value of x*y++ :200
```

```
/* program to demonstrate pre increment and decremenrt*/
#include<stdio.h>
#include<conio.h>
void main( )
{
        int a,b,x,y;
        clrscr( );
        a=4;b=5;x=10;y=20;
        printf("value of a :%d\n",a);
        printf("value of ++a :%d\n",++a);
        printf("new value of a :%d\n",a);
        printf("value of b :%d\n",b);
        printf("value of --b :%d\n",--b);
        printf("value of b :%d\n",b);
        printf("value of x*++y :%d\n",x*++y);
        getch( );
}
```

Output:

```
value of a :4
value of ++a :5
new value of a :5
value of b :5
value of --b :4
value of b :4
value of x*++y :210
```

6. Bit wise operators:

The smallest element in the memory on which we are able to operate is a byte (8 bits). C supports several bit wise operators. These permit the programmer to access and manipulate individual bits within a piece of data. The various bit wise operators available in C. These operators can operate on integers and characters but not on float.

Operator	Meaning
&	Bitwise AND
|	Bitwise OR
^	Bitwise exclusive OR
<<	Shift left
>>	Shift right
~	Ones complement

a) Bit wise AND (&) operator: The operator is represented as ‘&’ and operates on two operands. While operating upon these operands they are compared on a bit-by-bit basis. (Both the operands must be of it type either chars or int).

The truth table for Bitwise & is:

INPUTS		OUTPUT
X	Y	Z
0	0	0
0	1	0
1	0	0
1	1	1

Ex: X=0000 0111(=7) Y=0000 1000(=8) Z=X&Y=0000 0000(=0)

b) Bit wise OR (|) operator: The operator is represented as ‘|’ and operates on two operands. While operating upon these two operands they are compared on a bit-by-bit basis. (Both the operands must be of same type either chars or ints).

The truth table for Bit wise OR is:

INPUTS		OUTPUT
X	Y	Z
0	0	0
0	1	1
1	0	1
1	1	1

Ex: X=0000 0111(=7) Y=0000 1000(=8) Z=X|Y=0000 1111(=15)

c) One’s complement (~): For a binary number if we take one’s complement all zero’s become 1 and one’s become 0’s.

Ex: X=0001; ~X=1110;

d) Bitwise Exclusive OR: The operator is represented as '^' and operates on two operands. While operating upon these two operands they are compared on a bit-by-bit basis. (Both the operands must be of same type either chars or ints).

The truth table for XOR is:

INPUTS		OUTPUT
X	Y	Z
0	0	0
0	1	1
1	0	1
1	1	0

e) Bitwise right shift (>>) and left shift (<<):

e.g.: x>>=2 means shift two bits to right in the binary value if x.

Shifting two bits right means the inputted number is to be divided by 2^s where s is number of shifts i.e., in short $y=n/2^s$ where n is number and s is number of positions to be shifted

e.g.: x<<=2 means shift two bits to left in the binary value if x.

Shifting two bits left means the inputted number is to be multiplied by 2^s in short $y=n/2^s$ where n is number and s is number of positions to be shifted

```
/* Program to demonstrate bit-wise operators*/
#include<stdio.h>
#include<conio.h>
void main( )
{
        int x,y,a,b,c,d;
        clrscr( );
        x=7;y=8;a=5;b=6;
        printf("value of x after shifting 2 bits right is : %d\n",x>>=2);
        printf("value of y after shifting 2 bits left is : %d\n",y<<=2);
        printf("c=a&b : %d\n",c=a&b);
        printf("d=a|b  :  %d\n",d=a|b);
        getch( );
}
```

Output:

```
value of x after shifting 2 bits right is : 1
value of y after shifting 2 bits left is : 32
c=a&b : 4
d=a|b   : 7
```

7. Conditional operator (or) ternary operator: The conditional operator is a pair of operators ("? and :") are sometimes called ternary operator since they take three operands, and it is condensed form of an *if-then-else* C statement.

The general form of Conditional Operator is:

exp 1? exp 2: exp 3;

The conditional operator works as follows: exprssion1 is evaluated first. If it is non zero(true), then the expression 2 is evaluated. If expression 1 is false, expression 3 is evaluated. Note that only one of the expressions is evaluated.

```
Ex:     y=(x>5? 3:4) is equivalent to     if(x>5)
                                          then    y=3;
                                          else     y=4;
```

```
/* Program to demonstrate conditional operator*/
#include<stdio.h>
#include<conio.h>
void main( )
{        clrscr( );
         3>2?printf("TRUE"):printf("FALSE");
         getch( );
}
```

Output: TRUE

8. Special operators:

i) Comma operator (,): The comma (,) operator permits two different expressions to appear in situation where only one expression would ordinarily be used. The expressions are separated by comma operator.

Ex: c= (a=10, b=20,a+b);

Here firstly value 10 is assigned to a followed by this 20 is assigned to b and then the result of a+b is assigned to c.

ii) Sizeof operator: The size of operator returns the number of bytes the operand occupies in memory. The operand may be a variable, a constant or a data type qualifier.

Ex: sizeof(int) is going to return 2

iii) Address of operator (&): The address of operator (&) returns the address of the variable. The operand may be a variable, a constant.

Ex: m=&n;

Here address of n is assigned to m. This m is not a ordinary variable, it is a variable which holds the address of the other variable (i.e., pointer variable).

iv)Value at address operator (*): The value at address operator (*) returns the value stored at a particular address. The 'value at address' operator is also called an 'indirection' operator.

Ex: x=*m;

The value at address of m is assigned to x. Here m is going to hold the address.

```
/* Program to demonstrate special operators*/
#include<stdio.h>
#include<conio.h>
void main( )
{
        int x=8,*m;
        clrscr( );
        m=&x;
        printf("value of x :%d\n",x);
        printf("value of x by using pointer :%d\n",*m);
        printf("address of x : %u\n",&x);
        printf("address of x using m : %u\n",m);
        printf("address of m : %u\n",&m);
        printf("sizeof(x) : %d\n",sizeof(x));
        getch( );
}
```

Output:

```
value of x :8
value of x by using pointer :8
address of x : 2293572
address of x using m : 2293572
address of m : 2293568
sizeof(x) : 4
```

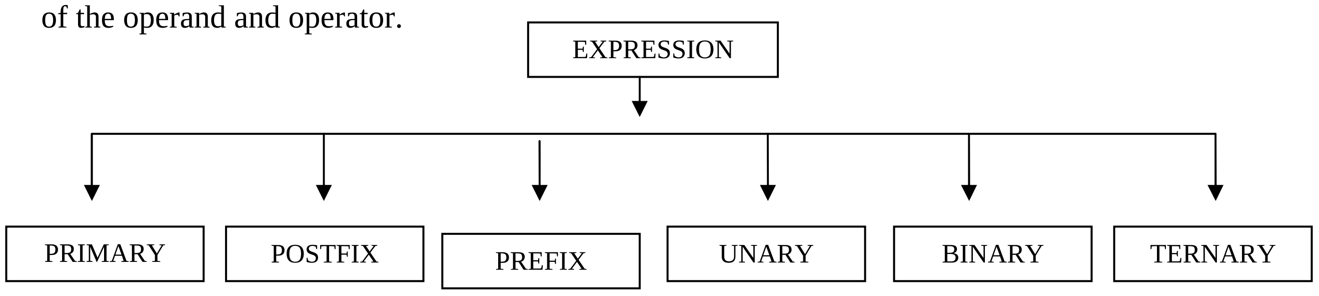
EXPRESSIONS:

An expression is a combination of variables, constants and operators arranged as per the syntax of the language.

Algebraic Expression	Equivalent C Expression
(a-b)(c+d)e	(a-b)*(c+d)*e
$4x^2$+8y+10	4*x*x+8*y+10
(a/b)+c	a/b+c
(ab/c)+d	(a*b/c)+d

A simple expression contains only one operator. For example 3+5 is a simple expression which yields a value 8. A complex expression contains more than one operator. An example of complex expression is 6+4*7/2.

An expression can be divided into six categories based on the number of operators, position of the operand and operator.



Primary Expressions: In C the operand in the primary expression can be a name, a constant or an parenthesized expression.

Postfix expressions: The postfix expression consists of one operand and one operator. For example A *Function call.* The function name is operand and parenthesis is the operator.

Prefix expressions: Prefix expressions consist of one operand and one operator, the operand comes after the operator. Examples of prefix expressions are prefix increment and prefix decrement i.e. ++a, -- a.

Unary expressions: An unary expression is like a prefix expression consists of one operand and one operator and the operand comes after the operator.

e.g. . +a, -a;

Binary expressions: Binary expressions are the combinations of two operands and an operator. Any two variables added, subtracted, multiplied or divided is a binary expression.

e.g: a+b; c*d;

Ternary expressions: Ternary expression is an expression which consists of a ternary operator pair ? :

e.g: exp1?exp2:exp3;

In the above example if exp1 is true exp2 is executed else exp3 is executed.

PRECEDENCE AND ASSOCIATIVITY: While executing an arithmetic statement which has two or more operators, we may have some problem about how exactly does it get executed. To answer these questions one has to understand the precedence of operators. The order of priority in which the operations are performed in an expression is called precedence. Associativity specifies the order in which the operators are evaluated with the same precedence. Associativity is of two ways i.e. left to right and right to left. The precedence and associativity of all operators is shown below.

Description	Operator	Precedence	Associativity
Function expression	( )	1	Left to Right
Array expression	[ ]		
Unary plus	+	2	Right to left
Unary minus	-		
Increment/Decrement	++/--		
Logical negation	!		
One’s complement	~		
Pointer reference	*		
Address of	&		
Size of an object	Sizeof		
Type cast (conversion)	(type)		
Multiplication	*	3	Left to Right
Division	/		
Modulus	%		
Addition	+	4	Left to Right
Subtraction	-		
Left shift	<<	5	Left to Right
Right shift	>>		
Less than	<	6	Left to Right
Less than or equal to	<=		
Greater than	>		
Greater than or equal to to	>=		
Equality	==	7	Left to Right
Not equal to	! =		
Bit wise AND	&	8	Left to Right
Bit wise XOR	^	9	Left to Right
Bit wise OR	|	10	Left to Right
Logical AND	&&	11	Left to Right
Logical OR	||	12	Left to Right
Conditional	? :	13	Right to Left
Assignment	=	14	Right to Left
	*= /= %=		
	+= -= &=		
	^= |= |		
	<<= >>=		
Comma operator	,	15	Left to Right

EVALUATIONS OF EXPRESSIONS:

Expressions are evaluated using an assignment statement of the form

Variable=expression;

The expressions are evaluated first and the result then replaces the previous value of the variable on the left hand side. All variables used in the expression must be assigned values before evaluation is attempted.

Examples of evaluation statement are:

x=b/c*a;   y=a-b/c+d;   z=a+b-c;

Rules for Expression Evaluation:

- First parenthesized sub expressions from left to right are evaluated.
- If parentheses are nested, the evaluation begins with the innermost sub-expression.
- When parentheses are used, the expressions within parentheses assume highest priority.
- The precedence rule is applied for evaluating sub-expressions.
- The associativity rule is applied when two or more operators of the same precedence level appear in a sub-expression.

TYPE CONVERSIONS:

C language permits mixing of constants and variables of different types in an expression, but during evaluation it adheres to very strict rules of type conversions. There are two types of type conversions.

i) Automatic type conversion or Implicit type conversion:

If the operands are of different types, the 'lower' type is automatically converted to the 'higher' type before the operation proceeds. The result is of the higher type.

Note: Some versions of C compilers automatically convert all floating-point operands to double precision.

The final result of an expression is converted to the type of variable on the left of the assignment operator before assigning the value to it.

However, the following changes are introduced during the final assignment:

i) float to int causes truncation of the fractional part.

ii) double to float causes rounding of digits.

iii) long int to int causes dropping of the excess higher order bits.

ii) Explicit type conversion (or) Type Casting:

C performs type conversions automatically. However, there are instances when we want to force a type conversion in a way that is different from the automatic type conversion. Consider, for example, the calculation of ratio of females to male in a town.

ratio=f_num/m_num;

Since f_num and m_num are declared as integers in the program, the decimal part of the result of the division would be lost and ratio would represent a wrong value. This problem can be solved by converting *locally* one of the variables to the floating point as shown below:

ratio= (float) f_num/m_num;

The operator (float) converts the f_num to floating point for the purpose of evaluation of the expression. Then using the rule of automatic type conversion, the division is performed in floating point mode, thus retaining the fractional part of result.

Note that in no way does the operator (float) affect the value of the variable f_num and also the type of the variable f_num remains as integer in the other parts of the program.

The process of such a local conversion is known as *casting* a value. The general form of a cast is: (type_name) expression;

Where *type_name* is one of the standard C data types. The *expression* may be a constant, variable or an expression.

Ex: Program showing the use of cast operator

```
#include<stdio.h>
#include<conio.h>
void main( )
{
        float sum;
        int n;
        clrscr( );
        sum=0;
        for(n=1;n<=10;++n)
        {
                sum=sum+1/(float)n;
                printf("%2d %6.4f\n",n,sum);
        }
}
```

Note: Casting can be use to round-off a given value. Consider the following statement:

x=(int) (y+0.5);   If y is 27.6, y+0.5 is 28.1 and on casting, the result becomes 28, the value that is assigned to x.

LIBRARY FUNCTIONS:

Library functions are pre-defined functions. A user can’t understand the internal working of the function and can only use those functions. These functions can’t be modified by the programmer.

‘C’ provides rich set of standard library functions whose definitions have been written and are ready to be used in the programs. The user must include the prototype declaration to use the standard library functions. The function prototypes for these functions are available in the general header files. Therefore, the appropriate header file for a standard function must be included at the beginning of our programs.

HEADER FILES: The entire ‘C’ library is divided into several files. Each such file is known as header file. Usually header files have an extension like **.h** to distinguish that from the ‘C’ program files.

The following table lists the names and description of some of the most commonly used functions along with the name of the standard header file to #include in order to access to each function.

Some mathematical library functions:

Function	standard header file	purpose: example	arguments	result
abs(x)	<stdlib.h>	Returns the absolute value of its integer argument:if x is -5 abs(x) is 5	int	int
ceil(x)	<math.h>	Returns the smallest integral value that is not less than x: if x is 45.23, ceil(x) is 46	double	double
cos(x)	<math.h>	Returns the cosine of angle x: if x is 0, cos(x)is 1.0	double (radians)	double
exp(x)	<math.h>	Returns e^x where e=2.71828……, if x is 1.0 exp(x) is 2.71828	double	double
fabs(x)	<math.h>	Returns the absolute value of its double argument : if x is -8.432, fabs(x) is 5.432	double	double
floor(x)	<math.h>	Returns the largest integral value that is not greater than x: if x is 45.23, floor(x) is 45	double	double
log(x)	<math.h>	Returns the natural logarithm of x for x>0.0: if x is 2.71828, log (x) is 1.0	double	double
log10(x)	<math.h>	Returns the base 10 logarithm of x for x>0.0: if x is 100 , log10(x) is 2.0	double	double
pow(x,y)	<math.h>	Returns x^y if x is 4.0 and y is 2.0 ,pow(x,y) is 16.0	double, double	double
sin(x)	<math.h>	Returns the sine angle of x: if x is 1.5708, sin(x) is 1.0	double (radians)	double
sqrt(x)	<math.h>	Returns the non negative square root of x for x>=0.0: if x is 2.25 , sqr Returns t(x) is 1.5	double	double
tan(x)	<math.h>	Returns the tangent of angle of x: if x is 0.0, tan(x) is 0.0	double (radians)	double

UNIT-II

Data Input and Output: Preliminaries, Single character input-getchar function, Single character output-putchar function, Entering input data-the scanf function, More about the scanf function, Writing output data-The printf function, More about the printf function, The gets and puts functions, Interactive(conversational) programming.

Preparing and running a complete C program: Planning a C program, Writing a C program, Error diagnostics, Debugging techniques.

Control statements: Preliminaries, Branching: if-else statement, Looping: The while statement, More looping: The do-while statement, Still more looping: The for statement, Nested control structures, The switch statement, Break statement, Continue statement, The comma operator, The goto statement.

Functions: A brief overview, Defining a function, Accessing a function, Function prototypes, Passing arguments to a function, Recursion

PRELIMINARIES:

An input/output function can be accessed from anywhere within a program simply by writing the function name, followed by a list of arguments enclosed in parentheses. The arguments represent data items that are sent to the function. Some input/output functions do not require arguments, though the empty parentheses must still appear.

The names of those functions that return data items may appear within expressions, as though each function reference were an ordinary variable (e.g., c = getchar ( ) ;), or they may be referenced as separate statements (e.g., scanf (. . .) ;). Some functions do not return any data items. Such functions are referenced as though they were separate statements (e.g., putchar (. . .) ;).

Most versions of C include a collection of header files that provide necessary information (e.g., symbolic constants) in support of the various library functions. Each file generally contains information in support of a group of related library functions. These files are entered into the program via an **#include** statement at the beginning of the program. **As** a rule, the header file required by the standard input/output library functions is called **stdio.h**

INPUT/OUTPUT OPERATIONS AND FUNCTIONS:

'C' language has no provision for either receiving data from any of the input devices such as keyboard etc., or for sending data to the output devices like monitor. Hence 'C' manages this with the help of standard library functions. A program using these functions should include the standard header file <stdio.h> at the beginning of the program using the following preprocessor directive.

All input/output operations in C are performed by special program units called input/output functions. The most common input/output functions are supplied as a part of the C input/output library to which we gain access through the preprocessor directive.

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

C provides several functions to support input and output statements. The input/output functions are classified into two categories as:

(a) Unformatted console I/O functions.
(b) Formatted console I/O functions

a) Unformatted Console I/O functions: Functions that accept a single argument (the argument must be data item representing a string or character) are concerned with unformatted I/O.

Type	:	Char	string
Input	:	getch( ),getche( ),getchar( )	gets( )
Output	:	putch ( ),putchar ( )	puts( )

i) getch( ) and getche( ): These functions will read a single character the instant it is typed without waiting for the key to be hit.

ii) getchar ( );It is very similar to the getche ( ) and getch ( ) functions echoing the character you type on the screen, but requires enter key to hit following the character you typed.

```
/*Example Program for getchar( ),getch( ),getche( ) */
#include<stdio.h>
#include<conio.h>
void main( )
{
        char ch;
        clrscr( );
        printf("\nEnter a Character:");
        ch=getchar();
        printf("\nResult is:%c",ch);
        printf("\nEnter a Character:");
        ch=getche( );
        printf("\nResult is:%c",ch);
        printf("\nEnter a Character:");
        ch=getch( );
        printf("\nResult is:%c",ch);
        getch( );
}
```

OUTPUT:

```
Enter a Character:K

Result is:K
Enter a Character:N
Result is:N
Enter a Character:
Result is:R
```

iii) putchar ( ) and putch ( ):These functions is used to write a character one at a time to the terminal.

```
/*Example Program for putchar( ), putch( ) */
#include<stdio.h>
#include<conio.h>
void main( )
{
        char ch;
        clrscr( );
        printf("\nEnter a Character:");
        ch=getchar();
        printf("\nResult 1:");
        putch(ch);
        printf("\nResult 2:");
        putchar(ch);
        getch( );
}
OUTPUT:
Enter a Character: K
Result 1:K
Result 2:K
```

iv) *gets( ) & puts( ):*

gets( ) receives a string which is an array of characters from the keyboard, puts( ) function works exactly opposite to gets( ) function i.e., prints the string on console.

```
/*Example program for gets( ),puts( )  */
#include<stdio.h>
#include<conio.h>
void main( )
 {
   char line[100];
   clrscr( );
   puts("Enter a string");
   gets(line);
   puts(line);
   getch( );
 }
OUTPUT:
Enter a string
this is c notes
this is c notes
```

b) Formatted Console I/O: Functions that accept strings as well as variable number of arguments to be displayed and read in a specified format are called as formatted I/O.

Type	:	char, int, float, string
Input	:	scanf ( )
Output	:	printf ( )

The above two functions are used to supply input from keyboard in a fixed format and obtain output in a specified format on the screen.

Formatted input function:

- Commonly used Standard input function is scanf function.

i) scanf function: This function can be used to read any combination of numerical values, single character and string.

The general format is:

scanf("control string",arg1,arg2,arg3,………..);

Where the control string refers to a string certain required formatting information, arg1, arg2,………..,arg n are arguments that represent the individual data items.

Ex: int n;

scanf("%d",&n);

Here the scanf function reads the value from user and store the value in the address of the variable 'n', which is represented as &n. Hence, when we use '&' symbol we refers to the address of the variable. scanf( ) has got its own limitations. Though both scanf( ) & gets are used for inputting a string, scanf( ) will not allow to input a string with blank spaces.

Within the control string, multiple character groups can be contiguous, or they can be separated by whitespace characters (i.e., blank spaces, tabs or newline characters). If whitespace characters are used to separate multiple character groups in the control string, then all consecutive whitespace characters in the input data will be read but ignored.

```
#include <stdio.h>
main( )
{
        char item[20]; int partno;float cost;
        . . . . .
        scanf("%s   %d   %f",item, &partno, &cost);
        . . . . .
}
```

Within the **scanf** function, the control string is **"%s %d %f ".** It contains three character groups. The first character group, **%s,** indicates that the first argument **(item)** represents a string. The second character group, **%d,** indicates that the second argument **(&partno)** represents a decimal integer value, and the third character group, **%f,** indicates that the third argument **(&cost)** represents a floating-point value.

scanf("%s%d%f", item, &partno, &cost);

with no whitespace characters in the control string. This is also valid, though the input data could be interpreted differently when using c-type conversions .

You can add a space before the %c. This is necessary because unlike other conversion specifiers, it doesn't skip whitespace. So when the user enters something like "10\n" as the age, the first scanf reads up to the end of 10. Then, the %c reads the newline. The space tells scanf to skip all the current whitespace before reading a character.

```
printf ("What is the customer's age? \n");
scanf("%d", &age);
printf ("Is the customer a student? (y/n) \n");
scanf(" %c", &edu);
printf ("What is the movies time? (in hours) \n");
scanf("%d", &time);
printf ("Is the movie 3-D? (y/n) \n");
scanf(" %c", &ddd);
```

problem of scanf( ) function while we reading character form keyboard:

Problem is: scanf( ) function reads all data and store them in buffer, after entering data generally we press newline character(enter) so scanf reads its as a one character and assigns to next scanf function.

Example:

```
#include<stdio.h>
#include<conio.h>
void main( )
{
    int x;
    char ch;
    clrscr( );
    printf("\nEnter value of x: ");
    scanf("%d",&x);
    printf("\nEnter any character: ");
    scanf("%c",&ch);
    printf("\nthe value of x is:%d",x);
    printf("\nThe value of ch is: %c",ch);
    getch( );
}
```

```
OUTPUT:
Enter value of x: 8


Enter any character:
the value of x is:8
The value of ch is:
```

Here character is not read from keyboard, it skips the line and executes the next statement because the new line character is assigned to ch variable.
To avoid this problem just use the space before the %c in scanf, it skips all the whitespaces before reading a character.

```
#include<stdio.h>
#include<conio.h>
main( )
{
    int x;
    char ch;
    clrscr(  );
    printf("\nEnter value of x: ");
    scanf("%d",&x);
    printf("\nEnter any characer: ");
    scanf("  %c",&ch);
    printf("\nthe value of x is:%d",x);
    printf("\nThe value of ch is: %c",ch);
    getch( );
}
OUTPUT:
Enter value of x: 8

Enter any characer: k

the value of x is:8
The value of ch is: k
```

Moreover, if the control string begins by reading a character-type data item, it is generally a good idea to precede the first conversion character with a blank space. This causes the scanf function to ignore any extraneous characters that may have been entered earlier (for example, by pressing the Enter key after entering a previous line of data).

MORE ABOUT SCANF( ):

- ✓ Note that the s-type conversion character applies to a string that is terminated by a whitespace character. Therefore, a string that ***includes*** whitespace characters cannot be entered using scanf ( ). We can use the **getchar( )** within a loop or **gets( )**.
- ✓ It is also possible to use the **scanf** function to enter such strings. For this control string is replaced by a sequence of characters enclosed in square brackets, designated **as** [. . .]. Whitespace characters may be included within the brackets, thus accommodating strings that contain such characters.
- ✓ When the program is executed, successive characters will continue to be read from the standard input device as long as each input character matches one of the characters enclosed within the brackets. The order of the characters within the square brackets need not correspond to the order of the characters being entered.
- ✓ Input characters may be repeated. The string will terminate, however, once an input character is encountered that does not match any of the characters within the brackets. **A** null character **(\0)** will then automatically be added to the end of the string.

```
#include<stdio.h>
#include<conio.h>
void main( )
{
        char line[80];
        clrscr( );
        printf("\nEnter any string \n");
        scanf("%[   ABCDEFGHIJKLMNOPQRSTUVWXYZ]",line);
        printf("\nThe string stored in line is: \n%s",line);
        getch( );
}
```

OUTPUT: Enter any string KNOWLEDGE IS POWER The string stored in line is: KNOWLEDGE IS POWER *the entire string will be assigned to the array **line** since the string is comprised entirely of uppercase letters and blank spaces*	 Enter any string Knowledge is power The string stored in line is: K *only the single letter **K** would be assigned to **line** , since the first lowercase letter (in this case, n**)** would be interpreted **as** the first character beyond the string*

A variation of this feature is to precede the characters within the square brackets by a ***Circumflex*** (i.e., ^). This causes the subsequent characters within the brackets to be interpreted in the opposite manner. Thus, when the program is executed, successive characters will continue to be read from the standard input device as long as each input character ***does not*** match one of the characters enclosed within the brackets.

If the characters within the brackets are simply the circumflex followed by a newline character, then the string entered from the standard input device can contain any ASCII characters except the newline character. Thus, the user may enter whatever he wishes and then press the **Enter** key. The **Enter** key will issue the newline character, thus signifying the end of the string.

```
#include<stdio.h>
#include<conio.h>
void main( )
{
        char line[80];
        clrscr( );
        printf("\nEnter any string \n");
        scanf("%[^\n]",line);
        printf("\nThe string stored in line is: \n%s",line);
        getch( );
}
OUTPUT:
Enter any string
we can enter any characters untill new line character

The string stored in line is:
we can enter any characters untill new line character
```

In the above example the scanf reads any characters until new line character is entered.
If scanf is written as **scanf(" %[^.]",line);** this reads characters until dot(.) the rest of characters will be ignored
For example if we run the above program with this scanf the output will be as follows:
Enter any string
knowledge. is power
The string stored in line is:
knowledge

It is possible to limit the number of such characters by specifying a maximum field width for that data item. To do so, an unsigned integer indicating the field width is placed within the control string, between the percent sign (%) and the conversion character.

The data item may contain fewer characters than the specified field width. However, the number of characters in the actual data item cannot exceed the specified field width. Any characters that extend beyond the specified field width will not be read. Such leftover characters may be incorrectly interpreted as the components of the next data item.

```
#include<stdio.h>
#include<conio.h>
void main( )
{
        int a,b,c;
        clrscr( );
        printf("\nEnter any three numbers:");
        scanf("%3d %3d %3d",&a,&b,&c);
        printf("a=%d\tb=%d\tc=%d",a,b,c);
        getch( );
}
OUTPUT:
Enter any three numbers:1 2 3
a=1     b=2     c=3
----------------------
Enter any three numbers:123 456 789
a=123   b=456   c=789
------------------------------------------
Enter any three numbers:123456789 12 4
a=123   b=456   c=789
-----------------------------------
Enter any three numbers:1234 567 89
a=123   b=4     c=567
```

If the control string contains multiple character groups without interspersed whitespace characters, then some care must be taken with c-type conversion. In such cases a whitespace character within the input data will be interpreted as a data item. To skip over such whitespace characters and read the next non whitespace character, the conversion group **%1s** should be used.

```
#include<stdio.h>
#include<conio.h>
void main( )
{
        char c1,c2,c3;
        clrscr( );
        printf("\nEnter three characters: ");
        scanf("%c%c%c",&c1,&c2,&c3);
        printf("c1=%c\tc2=%c\tc3=%c",c1,c2,c3);
        getch( );
}
OUTPUT;
Enter three characters: a b c
c1=a   c2=    c3=b
```

If the **scanf** function were written **as**

scanf("%c%ls%ls", &cl, &c2, &c3)

however, then the same input data would result in the following assignments:

cl = **a, c2** = **b, c3** = **c**

We could have written the **scanf** function **as**

scanf (" **%c %c %c", &cl, &c2, &c3);**

with blank spaces separating the **%c** terms, or we could have used the original **scanf** function but written the input data **as** consecutive characters without blanks; i.e., **abc.**

Unrecognized characters within the control string are expected to be matched by the same characters in the input data. Such input characters will be read into the computer, but not assigned to **an** identifier. Execution of the **scanf** function will terminate if a match is not found.

```
#include<stdio.h>
#include<conio.h>
void main( )
{
        int a,b;
        clrscr( );
        printf("\nEnter two numbers: ");
        scanf(" %d k %d",&a,&b);
        printf("a=%d\tb=%d",a,b);
        getch( );
}
OUTPUT:
Enter two numbers: 1 k 2
a=1     b=2
---------------------------------
Enter two numbers: 1 2
a=1     b=2293576 (garbage value)
```

If the input data consist of

1 k 2

*then the decimal integer **1** will be read in and assigned to a, the character **k** will be read in but subsequently ignored, and the value **2** will be read in and assigned to **b.***

*On the other hand, if the input were entered simply **as***

1 2

*then the **scanf** function would stop executing once the expected character **(k)** is not found. Therefore, **a** would be assigned the value **1** but **b** would automatically represent the garbage value.*

Formatted output function:

- Commonly used Standard output function is printf function.

i) printf function: Output data can be written from the computer on to a standard output device using the library function printf. This function can be used to output any combination of numerical values, single character and strings.

The general format is:

printf (“control string”, arg1, arg2, arg3………);

Where the control string refers to a string contains formatting information. When printf is called it opens the format string and reads a character at a time. If the character reads % or \ it does not print it but reads the character which is immediately followed by that % and \ have a special meaning to printf.

Format descriptors (or) Format Specifiers:

Format specifier	**Used for**
%d	for int and short int
%ld	for long int
%u	for unsigned int and unsigned short int
%lu	for long unsigned int
%f	for float
%e or %g	for float with an exponent
%lf	for double
%Lf	for long double
%c	for char
%s	for string
%o	for octal
%x	for hexa decimal

Escape Sequences or back space characters:

\n	new line
\t	horizontal tab
\v	vertical tabulator
\a	to beep the speaker
\’	single quote
\”	double quotes
\?	Question mark
\\	back slash
\0	NULL

Example program about printf

```
#include<stdio.h>
#include<conio.h>
void main ( )
{
     int a=8,b=1;
     char c='n';
     clrscr( );
     printf("%d  %c  %d",a,c,b);
     getch( );
}
here  the output will be : 8   n   1
If printf is written as  printf("%d  %c  %d",a,c,b);
then output will be  8n1
If printf is written as  printf("%d\n%c\n%d",a,c,b);
then output will be
8
n
1
```

A ***minimum*** field width can be specified by preceding the conversion character by an unsigned integer. If the number of characters in the corresponding data item is less than the specified field width, then the data item will be preceded by enough leading blanks to fill the specified field. **If** the number of characters in the data item exceeds the specified field width, however, then additional space will be allocated to the data item, so that the entire data item will be displayed. This is just the opposite of the field width indicator in the **scanf** function, which specifies a ***maximum*** field width.

In addition to the field width, the precision and the conversion character, each character group within the control string can include a flag, which affects the appearance of the output. The flag must be placed immediately after the percent sign (%). Some compilers allow two or more flags to appear consecutively, within the same character group

-	Data item is left justified within the field (blank spaces required to fill the minimum field width will be added after the data item rather than before the data item).
+	**A** sign (either + or -) will precede each signed numerical data item. Without this flag, only negative data items are preceded by a sign.

0 Causes leading zeros to appear instead of leading blanks. Applies only to data items that are right justified within a field whose minimum size is larger than the data item. (Note: Some compilers consider the zero flag to be a part of the field width specification rather than an actual flag. This assures that the 0 is processed last, if multiple flags are present.)

(blank space) A blank space will precede each positive signed numerical data item. This flag is overridden by the + flag if both are present.

# (with ***o***- and x-type conversion) Causes octal and hexadecimal data items to be preceded by ***0*** and 0x, respectively.

# (with **e-,** f-and g-type conversion) Causes a decimal point to be present in all floating-point numbers, even if the data item is a whole number. Also prevents the truncation of trailing zeros in g-type conversion.

- **Output of integer number:**

The format specification for printing an integer numbers is

%wd

Here w specifies the minimum field width for the output, However if a number is greater than the specified field width, the number will be printed in full, overriding the minimum specifications. d specifies that the value to be printed is an integer. The number is printed right-justified in the given field-width. Leading blanks will appear.

Ex: The table shows the output of number 9876 in differerent formats.

Format	**Output**					
printf("%d",9876);	9	8	7	6		
printf("%6d",9876);			9	8	7	6
printf("%2d",9876);	9	8	7	6		
printf("%-6d",9876);	9	8	7	6		
printf("%06d",9876);	0	0	9	8	7	6

Note: i)to print a number with left-justified by placing minus (-)sign after the % character.

ii) here – and 0 are called flags in printf statement.

The following example program illustrates the output of an integer in different formats

```
/*Output of integer numbers under various formats*/
#include<stdio.h>
#include<conio.h>
void main( )
{
        int i=3214;
        clrscr( );
        printf("i =%d\n",i);
        printf("i(%%3d)=%3d\n",i);
        printf("i(%%7d)=%7d\n",i);
        printf("i(%%-7d)=%-7d\n",i);
        printf("i(%%010d)=%010d\n",i);
        printf("i(%%.10d)=%.10d\n",i);
        printf("i(%%o)=%o\n",i);
        printf("i(%%x)=%x\n",i);
        printf("i(%%#o)=%#o\n",i);
        printf("i(%%#x)=%#x\n",i);
        printf("i(%%6d)=%6d\n",-i);
        getch( );
}
OUTPUT:
 i =3214
i(%3d)=3214
i(%7d)=   3214
i(%-7d)=3214
i(%010d)=0000003214
i(%.10d)=0000003214
i(%o)=6216
i(%x)=c8e
i(%#o)=06216
i(%#x)=0xc8e
i(%6d)=   -3214
```

- **Output of real numbers:**

The real numbers may be displayed in decimal notation using the following format specification. **%w.p f**

Here w indicates the minimum number of positions that are to be used for display the value and p indicates the number of digits to be displayed after the decimal point (precision). The value is rounded to p decimal places and printed. The default precision is 6 decimal places.

We can also display a real number in exponential notation by using the specification.

%w.p e

The following example program illustrates the output of an real number in different formats

Ex: The table shows the displaying a real number y=98.7654 in different formats.

Format	**Output**
printf("%7.4f",y);	`98.7654`
printf("%7.2f",y);	`  98.77`
printf("%-7.2f",y);	`98.77  `
printf("%f",y);	`98.7654`
printf("%10.2e",y);	`  9.88e+01`
printf("%11.4e",-y);	`-9.8765e+01`
printf("%-10.2e",y);	`9.88e+01  `
printf("%e",y);	`9.876540e+01`

```
/*Output of real numbers in various formats*/
#include<stdio.h>
#include<conio.h>
void main( )
{
        float i=98.7654;
        clrscr( );
        printf("i(%%f)=%f\n",i);
        printf("i(%%f)=%f\n",-i);
        printf("i(%%+.0f)=%+.0f\n",i);
        printf("i(%%-.0f)=%-.0f\n",i);
        printf("i(%%8.2f)=%8.2f\n",i);
        printf("i(%%6.8f)=%6.8f\n",i);
        printf("i(%%2.2f)=%2.2f\n",i);
        printf("i(%%10.2e)=%10.2e\n",i);
        printf("i(%%09.2f)=%09.2f\n",i);
        printf("i(%%9.2f)=%9.2f\n",i);
        printf("i(%%012.2f)=%012.2f\n",i);
        printf("i(%%12.2f)=%12.2f\n",i);
        printf("i(%%8.2f)=%8.2f\n",i);
        printf("i(%%#10.0f)=%#10.0f\n",i);
        printf("i(%%e)=%e\n",i);
        printf("i(%%*.*f82)=%*.*f",8,2,i);
        getch( );
}
```

```
OUTPUT:
i(%f)=98.765404
i(%f)=-98.765404
i(%+.0f)=+99
i(%-.0f)=99
i(%8.2f)=   98.77
i(%6.8f)=98.76540375
i(%2.2f)=98.77
i(%10.2e)= 9.88e+001
i(%09.2f)=000098.77
i(%9.2f)=    98.77
i(%012.2f)=000000098.77
i(%12.2f)=       98.77
i(%8.2f)=   98.77
i(%#10.0f)=       99.
i(%e)=9.876540e+001
i(%*.*f82)=   98.77
```

- **Output of characters and strings:** The characters and strings may be displayed by using the following format specification.

%w.pf

Here w specifies the field for display and p instructs that the first p characters of the string are to be displayed. The display is right justified. The following example program illustrates the output of characters and strings in different formats

```
/*printing of characters and strings in various formats*/
#include<stdio.h>
#include<conio.h>
void main( )
{
        char name[50]="KNREDDY SK INFOTECH",ch='S';
        clrscr( );
        printf("ch=%c\n",ch);
        printf("ch=%3c\n",ch);
        printf("ch=%6c\n",ch);
        printf("name=%s\n",name);
        printf("name=%15s\n",name);
        printf("name=%*s\n",30,name);
        printf("name=%20.10s\n",name);
        printf("name=%-20.10s\n",name);
        printf("name=%.7s\n",name);
        getch( );
}
```

OUTPUT:

```
ch=S
ch=  S
ch=     S
name=KNREDDY SK INFOTECH
name=KNREDDY SK INFOTECH
name=           KNREDDY SK INFOTECH
name=          KNREDDY SK
name=KNREDDY SK
name=KNREDDY
```

We can specify variable field width in the printf statement as:

printf ("%*d\n", i, num);   /* field width of i characters */

```
#include<stdio.h>
#include<conio.h>
void main( )
{
        int i,num;
        clrscr( );
        printf("\nEnter a number :");
        scanf("%d",&num);
        for(i=1;i<=num;i++)
                printf("%0*d\n",i,i);
        i=i-2;
        for( ;i>=1;i--)
                printf("%0*d\n",i,i);
        getch( );
}
```

OUTPUT:

```
Enter a number : 4
1
02
003
0004
003
02
1
```

INTERACTIVE (CONVERSATIONAL) PROGRAMMING

- ✓ Many modern computer programs are designed to create an interactive dialog between the computer and the person using the program (the "user").
- ✓ These dialogs usually involve some form of question-answer interaction, where the computer asks the questions and the user provides the answers, or vice versa.
- ✓ In **C,** such dialogs can be created by alternate use of the scanf and printf functions.
- ✓ The printf function is used for entering data (to create the computer's questions) and for displaying results. On the other hand, scanf is used only for actual data entry.

EG: Program for Averaging Student Exam Scores
The following interactive C program reads in a student's name and three exam scores, and then calculates an average score.
The data will be entered interactively, with the computer asking the user for information and the user supplying the information in a free format, as requested.
Each input data item will be entered on a separate line. Once all of the data have been entered, the computer will compute the desired average and write out all of the data (both the input data and the calculated average).

```
/* sample interactive program */
#include<stdio.h>
#include<conio.h>
void main( )
{
    char name[20];
    float score1 , score2, score3, avg;
    clrscr( );
    printf("P1ease enter your name: " );     /* enter name */
    scanf("%[^\n]" , name);
    printf("\nP1ease enter the first score: " ); /* enter 1st score */
    scanf(" %f",&score1);
    printf("\nP1ease enter the second score: " ); /* enter 2nd score */
    scanf(" %f",&score2);
    printf("\nP1ease enter the third score: " ); /* enter 3rd score */
    scanf(" %f",&score3);
    avg = (score1+score2+score3)/3;      /* calculate avg */
    printf("\n\nName: %s\t", name); /* write output */
    printf("\nScore1: %-5.1f", score1);
    printf("\nScore2: %-5.1f", score2);
    printf("\nScore3: %-5.1f", score3);
    printf("\nAverage: %-5.1f", avg);
    getch( );
}
OUTPUT:
P1ease enter your name: KNREDDY
P1ease enter the first score: 88
P1ease enter the second score: 76
P1ease enter the third score: 84
Name: KNREDDY
Score1: 88.0
Score2: 76.0
Score3: 84.0
Average: 82.7
```

PREPARING AND RUNNING A COMPLETE C PROGRAM

PLANNING A C PROGRAM

- It is essential that the overall program strategy be completely mapped out before any of the detailed programming actually begins.
- The general program logic is written, without being concerned with the syntactic details of the actual instructions.
- This approach is generally referred to as "top-down" programming.
- For larger programs this details are added repeatedly.
- In the initial stages of program development the amount of C is minimal, consisting only of major program components, such as function headings, function references, braces defining compound statements, and portions of control statements describing major program structures. Additional detail is then provided by descriptive English material which **is** inserted between these elements, often in the form of program comments. The resulting outline is usually referred to as ***pseudocode.***

Consider an example how a program is planned for finding Compound Interest

P -principal n- number of years r- rate of interest per year

Compound interest can be calculated by using the formula $\boxed{F = P(1+i)^n}$

where *F* represents final amount (p-principal + interest*)* and *i* is the decimal representation of the interest rate; i.e., *i* = r/100 (for example, an interest rate of *r = 5%* would correspond to *i* = 0.05).

The program will be based upon the following general outline.

1. Declare the required program variables.
2. Read in values for the principal (P), the interest rate *(r)*and the number of years *(n).*
3. Calculate the decimal representation of the interest rate *(i),*using the formula *i* = r/100
4. Determine the total amount *(F)*using the formula *F = P (l +i)"*
5. Display the calculated value for *F.*

Here is the program outline in the form of pseudocode.

```
/* compound interest calculations */
main( )
{
/* declare the program variables */
/* read in values for P, r and n */
/* calculate a value for i*/
/* calculate a value for F */
/* display the calculated value for F */
}
```

Each of these steps is very simple. Some steps require more detail before they can actually be programmed.

For example, the data input step will be carried out interactively. This will require some dialog generated by pairs of printf and scanf statements, and functions to evaluate exponentiation.

After refinement of the above pseudo code it is as follows:

```
/* compound interest calculations */
main( )
{
/* declare p, r, n, i and f to be floating-point variables */
/* write a prompt for p and then read in its value */
/* write a prompt for r and then read in its value */
/* write a prompt for n and then read in its value */
/* calculate i= r/100 */
/ * calculate f = p (1 + i )^n as:
        f = p*pow((l+i),n)      where pow is a library function for exponentiation */
/* display the value for f, with an accompanying label */
```

WRITING A C PROGRAM

- Once an overall program strategy has been formulated and a program outline has been written, then the detailed C program can be written by translating each step of the program outline (or each portion of the pseudocode) into one or more equivalent C instructions.
- In writing a complete C program , arranging the individual declarations and statements in the right order and then punctuating them correctly.
- Including certain additional features that will improve the readability of the program and its resulting output. These features include the logical sequencing of the statements, the use of indentation and whitespace, the inclusion of comments and the generation of clearly labeled output.
- It is important that the program statements be selected and sequenced in the most effective manner.
- The use of indentation is closely related to the sequencing of groups of statements within a program. Whereas sequencing affects the order in which a group of operations is carried out, indentation illustrates the subordinate nature of individual statements within a group.
- In addition, blank lines are sometimes used to separate related groups of statements.
- Comments should always be included within a C program. If written properly, comments can provide a useful overview of the general program logic. Generally, the comments need not be extensive;
- Comments can be of great use to the original programmer as well as to other persons trying to read and understand a program, since most programmers do not remember the details of their own programs over a period of time. This is especially true of programs that are long and complicated.
- Another important characteristic of a well-written program is its ability to generate clear, legible output. Two factors contribute to this legibility. The first is labeling of the output data, The second is the appearance of some of the input data along with the output, so that each instance of program execution (if there are more than one) can be clearly identified.
- In an interactive environment the input data is displayed on the screen at the time of data entry, during program execution.
- When executing an interactive program, the *user* (someone other than the programmer) may not know how to enter the required input data. For example, the user may not know what data items are required, when the data items should be entered, or the order in which they should be entered. Thus a well-written interactive program should generate ***prompts*** at appropriate times during the program execution in order to provide this information.

```
/*compound interest problem */
#include<stdio.h>
#include<conio.h>
#include<math.h>
void main( )
{
        float p,r,n,i,f;
        clrscr( );
        /* read input data (including prompts) */
        printf("\nP1ease enter a value for principal(P): " ) ;
        scanf ("%f",&p);
        printf("\nP1ease enter a value for rate of interest( r ) : " ) ;
        scanf ("%f" , &r);
        printf("\nP1ease enter a value for the number of years (n): " ) ;
        scanf (" %f",&n);
        /* calculate i,then f */
        i= r/100;
        f = p * pow((1 + i ),n);
        /* display the output */
        printf("\nThe final amount  (F) is : %.2f " ,f );
        getch( );
}
```

OUTPUT:

```
P1ease enter a value for principal(P): 1000

P1ease enter a value for rate of interest( r ) : 10

P1ease enter a value for the number of years (n): 2

The final amount (F) is: 1210.00
```

ERROR DIAGNOSTICS

- Programming errors are detected when the program is compiling or execute.
- There are three kinds of programming errors:
    - syntactic errors
    - run time errors or execution errors
    - semantic errors or logical errors
- The presence of *syntactic* (or *grammatical)* errors is visible once the program is compiled. Some particularly common errors of this type are improperly declared variables, a reference to an undeclared variable, incorrect punctuation, etc.
- Most C compilers will generate *diagnostic messages* when syntactic errors have been detected during the compilation process. These diagnostic messages are not always straightforward in their meaning and they may not correctly identify where the error occurred.
- They are helpful in identifying the nature and the approximate location of the errors.
- If a program includes several different syntactic errors, they may not all be detected on the first pass through the compiler. Thus, it may be necessary to correct some syntactic errors before others can be found. This process could repeat itself through several cycles before all of the syntactic errors have been identified and corrected.

  Consider the following program with some errors:

```
/*compound interest problem */
#include<stdio.h>
include<conio.h>
#include<math.h>
void main( )
{    float p,r,n,i,f;
     clrscr( );
     /* read input data (including prompts) */
     printf("\nP1ease enter a value for principal(P): " ) ;
     scanf ("%f",&p);
     printf("\nP1ease enter a value for rate of interest( r ) : ) ;
     scanf ("%f" , &r);
     printf("\nP1ease enter a value for the number of years (n): " ) ;
     scanf (" %f",&n)
     /* calculate i,then f */
     i= r/100;
     f = p * pow((1 + i ),n;
     /* display the output /*
     printf("\nThe final amount  (F) is : %.2f " ,f );
     getch( );
}
```

This program contains five different syntactic errors. The errors are **as** follows:

1. The second **include** statement does not begin with a # sign.

2. The control string in the second **printf** statement does not have a closing quotation mark

3. The last scanf statement does not end with a semicolon.

4. The assignment statement for f contains unbalanced parentheses.

5. The last comment closes improperly (it ends with / * instead of * /).

After compiling the above program we get the following errors.

```
Compiling COMPOUND.C:
Error COMPOUND.C 3: Declaration syntax error
Error COMPOUND.C 12: Unterminated string or character constant
Error COMPOUND.C 13: Function call missing )
Error COMPOUND.C 17: Statement missing ;
Error COMPOUND.C 18: Function call missing )
Error COMPOUND.C 24: Unexpected end of file in comment started on line 19
Error COMPOUND.C 18: Compound statement missing }
```

- The first message specifies that # is missed in line 3 and second message says that there is missing double quotes in the printf statement in line 12.

When these two errors are corrected we get the following errors.

```
Compiling COMPOUND.C:
Error COMPOUND.C 17: Statement missing ;
Error COMPOUND.C 18: Function call missing )
Error COMPOUND.C 24: Unexpected end of file in comment started on line 19
Error COMPOUND.C 18: Compound statement missing }
```

- The first error message refers to the missing semicolon at the end of the last scanf statement (which actually occurs in line 15, not line 17). The second message refers to the missing left parenthesis in second assignment statement (line **18).**
- When these remaining errors were corrected, the program compiled correctly and began to execute

Another type of error that is quite common is the ***execution* error**. Execution errors occur during program execution, after a successful compilation. For example, some common execution errors are a *numerical overflow* of *underflow* (exceeding the largest or smallest permissible number that can be stored in the computer), division by zero, attempting to compute the logarithm or the square root of a negative number, etc. Diagnostic messages will often be generated in situations of this type, making it easy to identify and correct the errors. These diagnostics are sometimes called *execution* messages or *run-time* messages, to distinguish them from the *compilation* messages.

- The syntactic errors and execution errors usually produce error messages when compiling or executing a program.
- Syntactic errors are relatively easy to find and correct, even if the resulting error messages are unclear. Execution errors are difficult to identify.
- When an execution error occurs, we must first determine its location *(where* it occurs) within the program. Once the location of the execution error has been identified, the source of the error *(why* it occurs) must be determined.
- Closely related to execution errors are **semantic or Logical errors**. Here the program executes correctly, carrying out the programmer's wishes, but the programmer has supplied the computer with instructions that are logically incorrect. Logical errors can be very difficult to detect, since the output resulting from a logically incorrect program may appear to be error-free. Moreover, logical errors are often hard to locate even when they are known to exist (as, for example, when the computed results are obviously incorrect).

```
/* real roots of a quadratic equation */
#include <stdio.h>
#include<conio.h>
#include <math.h>
void main( )
{
        float a, b, c, d, xl , x2;
        clrscr( );
        /* read input data */
        printf("a =");
        scanf( "%f", &a);
        printf("b =");
        scanf( "%f",&b);
        printf("c=");
        scanf("%f",&c);
        /* carry out the calculations */
        d = sqrt(b * b - 4 * a * c ) ;
        xl =( - b + d)/(2 * a);
        x2 =( - b - d)/(2 * a);
        /* display the output */
        printf("\n xl =%e \t x2 = %e", xl , x2);
        getch( );
}
```

The above program is compiled with no errors (syntax errors). When this program is executed with the input values 1e-30   1e10   1e36   it generates a runtime error (floating point overflow)

DEBUGGING TECHNIQUES

- There are different methods for finding the location of execution errors and logical errors within a program. Such methods are generally referred to as ***debugging techniques***.
    - Error isolation
    - Tracing
    - Watch values
    - Breakpoints
    - Stepping

Error isolation

- Error isolation is useful for locating an error resulting in a diagnostic message.
- If the general location of the error is not known, it can frequently be found by temporarily deleting a portion of the program and then rerunning the program to see if the error disappears. The temporary deletion is accomplished by surrounding the instructions with comment markers (/ * and * /), causing the enclosed instructions to become comments.
- If the error message then disappears, the deleted portion of the program contains the source of the error.

```
#include <stdio.h>
#include<conio.h>
#include <math.h>
void main( )
{
            float a, b, c, d, xl , x2;
            clrscr( );
            /* read input data */
            printf("a =");
            scanf( "%f", &a);
            printf("b =");
            scanf( "%f",&b);
            printf("c=");
            scanf("%f",&c);
            /* carry out the calculations */
/***************************** error isolation
            d = sqrt(b * b - 4 * a * c ) ;
            xl =( - b + d)/(2 * a);
            x2 =( - b - d)/(2 * a);
********************* end of error isolation*/
            /* display the output */
            printf("\n xl =%e \t x2 = %e", xl , x2);
            getch( );
}
```

When the altered program was executed with the same three input values, the error message did not appear (though the displayed values for **x1** and **x2** did not make any sense). Thus, it is clear that the source of the original error message lies in one of these three statements.

- **A** closely related technique is that of inserting several unique printf statements, such as

  printf ("Debugging - line 1\ n ") ;

  printf ("Debugging - line 2 \ n ") ;

  etc. at various places within the program.
- When the program is executed, the debug messages will indicate the approximate location of the error. Thus, the source of the error will lie somewhere between the last printf statement whose message *did* appear, and the first printf statement whose message *did not* appear.

```
#include <stdio.h>
#include<conio.h>
#include <math.h>
void main( )
{
            float a, b, c, d, xl , x2;
            clrscr( );
            /* read input data */
            printf("a =");
            scanf( "%f", &a);
            printf("b =");
            scanf( "%f",&b);
            printf("c=");
            scanf("%f",&c);
            /* carry out the calculations */
            printf("Debugging line1\n");
            d = sqrt(b * b - 4 * a * c ) ;
            printf("Debugging line2\n");
            xl =( - b + d)/(2 * a);
            printf("Debugging line3\n");
            x2 =( - b - d)/(2 * a);
            /* display the output */
            printf("\n xl =%e \t x2 = %e", xl , x2);
            getch( );
}
```

When the program was executed, again using the same three input values, all three debug messages appeared; i.e.,

```
Debugging - Line 1
Debugging - Line 2
Debugging - Line 3
```

Hence, we conclude that the overflow occurred in the last assignment statement, since this statement follows the third printf statement.

Tracing

- *Tracing* involves the use of printf statements to display the values assigned to certain key variables, or to display the values that are calculated internally at various locations within the program.
- This information serves several purposes. For example, it verifies that the values actually assigned to certain variables really are (or are not) the values that should be assigned to those values.
- With these values we are able to identify a particular place where things begin to go wrong because the values generated will be obviously incorrect.

```
#include <stdio.h>
#include<conio.h>
#include <math.h>
void main( )
{
        float a, b, c, d, xl , x2;
        clrscr( );
        /* read input data */
        printf("a =");
        scanf( "%f", &a);
        printf("b =");
        scanf( "%f",&b);
        printf("c=");
        scanf("%f",&c);
        /* carry out the calculations */
        d = sqrt(b * b - 4 * a * c ) ;
        printf( " a = %e \nb = %e \nc = %e d = %e\n", a, b, c, d);     /* tracing statement */
        printf( " - b + d = %e\n", ( - b + d ) ) ;                    / * tracing statement */
        printf( " - b - d = %e\n", ( - b - d));                        / * tracing statement */
        xl =( - b + d)/(2 * a);
        x2 =( - b - d)/(2 * a);
        /* display the output */
        printf("\n xl =%e \t x2 = %e", xl , x2);
        getch( );
}
```

Execution of this program resulted in the following output:

```
a = 1.000000e-30   b = 1.000000e+l0      c = 1.000000e+36     d = 1.000000e+10
-b + d = 0.000000e+00                    -b - d = -2.000000e+10
```

From these results we can now determine that the value of x2 should be

x2 = (- b - d) / (2 * a) = (-2.000000e+10) / (2 x 1.000000e-30) = -1.000000e+40

The resulting value, -1 .000000e+40, exceeds (in magnitude) the largest floating-point number that can be stored within the computer's memory

Most contemporary C compilers include an ***interactive debugger,*** which provides the ability to set ***watch values*** and ***breakpoints,*** and allows ***stepping*** through a program one instruction at a time.

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 monitoring a few selected watch values, we can often determine where the program begins to generate incorrect or unexpected values.
- In Turbo C++, watch values can be defined by selecting **Add Watch** from the Debug menu , and then specifying one or more variables or expressions in the resulting dialog box. The watch values will then be displayed within a separate window as the program executes.

Breakpoints

- **A *breakpoint*** is a temporary stopping point within a program. Each breakpoint is associated with a 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 then 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 breakpoint as the program executes.
- To set a breakpoint in Turbo C++, select **Add Breakpoint** from the Debug menu, and then provide the requested information in the resulting dialog box.
- Or, select a particular line within the program and designate it a breakpoint by pressing function key F5. The breakpoint may later be disabled by again pressing F5. (Function key F5 is called a "toggle" in this context, since it turns the breakpoint on or off by successively pressing the key.)

Stepping

- ***Stepping*** refers to the execution of one instruction at a time, typically by pressing a function key to execute each instruction.
- In Turbo C++, for example, stepping can be carried out by pressing either function key F7 or F8. (F8 steps over subordinate functions, whereas F7 steps through the functions.) By stepping through an entire program, you can determine which instructions produce erroneous results or generate error messages.
- Stepping is often used with watch values, allowing you to trace the entire history of a program as it executes. This allows us to determine which instructions generate erroneous results.

CONTROL STATEMENTS

A C program is a set of statements which are normally executed sequentially in the order in which they appear. But sometimes there are a number of situations, where we may have to change the order of execution of statements based on certain conditions, or repeat a group of statements until certain specified conditions are met. This involves a kind of decision making statements (Control statements) to see whether a particular condition has occurred or not, then based on the result of those statements will directs the compiler to execute certain statements accordingly.

In C language, control statements are classified into three categories as:

1. Selection (or) Decision Control Statements
2. Loop (or) Iterative Control Statements
3. Branch (or) Jump Control Statements

1. Selection (or)branching (or) Decision Control Statements: The following are the selection statements

   a) Simple if statements

   b) if else statement

   c) Nested if-else statement

   d) else-if statement

   e) Switch statement.

2. Iterative statements (looping): The iterative statements allow a set of instruction to be performed until a certain condition is reached. C provides three different types of loops namely.

   a) while loop

   b) do-while loop

   c) for loop

3. Branch (or) Jump Control Statements: Branch control statements are used to transfer the control from one place to another place. C language provides three branch control statements.

   i) break statement

   ii) continue statement

   iii) goto statement

Branching statements:

a) Simple if statement: The if statements is used to specify conditional execution of program statements, or a group of statements enclosed in braces.

The general format is:

```
if(condition)
{
        Block Statements;
}
StatementsX;
```

Here,

- First condition is evaluated. It produces either TRUE or FALSE.
- If the condition outcome is TRUE, then Block Statements are executed by the compiler. After executing the block statements, control reaches to Statements-X.
- If the condition outcome is FALSE, then Block Statements are skipped by the compiler and control reaches to Statements-X.
- Block Statements may be either simple or compound statements. If the statements are a simple statement, pair of braces is optional.

Ex:- */* Program to print absolute value of the given integer*/*

```
#include<stdio.h>
#include<conio.h>
void main ( )
{
        int x;
        clrscr( );
        printf (“enter any integer number :”);
        scanf (“%d”, &x);
        if(x<0)
                x=-x;
        printf (“absolute  value of the given number is :%d\n”, x);
}
```

Output:

```
Enter any integer number:-10
Absolute value of the given number is: 10
```

b) if–else statement: The if statement allows conditional execution of a group of statements. However, there are situations when there are two groups of statements and it is desired that one of them executed if some condition is true and the other be executed if the condition is false. In such situation, we make use of if-else statement.

The general format:

```
if(condition)
{
        Block-I Statements;
}
else
{
        Block-II Statements;
}
Statements-X;
```

Here,

- First condition is evaluated. It produces either TRUE or FALSE.
- If the condition outcome is TRUE, then Block-I Statements are executed by the compiler. After executing the block-I statements, control reaches to Statements-X.
- If the condition outcome is FALSE, then Block-II Statements are executed by the compiler. After executing the block-II statements, control reaches to Statements-X.

Ex: */* Program to find out the accepted number is positive or negative*/*

```
#include <stdio.h>
#include<conio.h>
void main( )
{
        int x;
        printf("enter any integer number");
        scanf("%d",&x);
        if(x<0)
                printf("the given number is positive :");
        else
                printf("the given number is negative:");
}
```

Output

```
Enter any integer number:-12
The given number is negative.
```

```
/*PROGRAM TO FIND MAXIMUM OF TWO NUMBERS*/
#include<stdio.h>
#include<conio.h>
void main( )
{
        int num1,num2;
        clrscr( );
        printf("Enter any two numbers\n");
        scanf("%d%d",&num1,&num2);
        if(num1>num2)
        {
                printf("%d is maximum of %d,%d",num1,num1,num2);
        }
        else
        {
                printf("%d is maximum of %d,%d",num2,num1,num2);
        }
        getch( );
}
OUTPUT:
Enter any two numbers
20
40
40 is maximum of 20, 40
```

```
/*PROGRAM TO FIND GIVEN NUMBER IS EVEN OR ODD*/
#include<stdio.h>
#include<conio.h>
void main( )
{
        int n;
        clrscr( );
        printf("Enter the number : ");
        scanf("%d",&n);
        if (n%2==0)
        {
                printf("\nNumber is Even");
        }
        else
        {
                printf("\nNumber is Odd");
        }
        getch( );
}
OUTPUT:
Enter the number : 120

Number is Even
```

c) Nested if –else statements: When a series of decisions are involved we may have to use more than one if else statements in nested form (i.e., an *if* statement is placed within another *if* statement) as follows:

The general format:

```
if(condition1)
{
        if(condition2)
        {
                Block-I Statements;
        }
        else
        {
                Block-II Statements;
        }
}
else
{
        Block-III Statements;
}
Statements-X;
```

Here,

- First condition1 is evaluated. It produces either TRUE or FALSE.
- If the condition1 outcome is TRUE, then control enters into condition2 section. Condition2 produces either TRUE or FALSE.
  - If Condition2 is TRUE, then Block-I Statements are executed by the compiler. After executing the block-I statements, control reaches to Statements-X.
  - If Condition2 is FALSE, then Block-II Statements are executed by the compiler. After executing the block-II statements, control reaches to Statements-X.

If the condition1 outcome is FALSE, then Block-III Statements are executed by the compiler. After executing the block-III statements, control reaches to Statements-X.

Ex: */* Program to find out the maximum number from the given three numbers*/*

```
#include<stdio.h>
#include<conio.h>
void main( )
{
        int x,y,z max;
        clrscr( );
        printf("enter first number");
        scanf("%d",&x);
        printf("enter second number");
        scanf("%d",&y);
        printf("enter third number");
        scanf("%d",&z);
        if(x<y)
        {       if (y<z)
                        max=z;
                else
                        max=y;
        }
        else
        {
                if(x<z)
                        max=z;
                else
                        max=x;
        }
printf("max number of %d%d%d is :%d \n",x,y,z,max);
}
```

Output:

```
Enter first number:8
Enter second number:23
Enter third number:9
Max number of 8 23 9 is :23
```

d) else – if ladder :

Here the conditions are evaluated from the top to bottom. As soon as a true condition is found the statements associated with it is executed and the rest of the ladder is by passed. The last else handles the defaults case.

The general format

```
if(condition1)
{
        Block-I Statements;
}
else if(condition2)
{
        Block-II Statements;
}
.
.
.
else if(conditionn)
{
        Block-n Statements;
}
else
{
        ElseBlock Statements;
}
Statements-X;
```

Here,

- First condition1 is evaluated. It produces either TRUE or FALSE.
- If the condition1 outcome is TRUE, then Block-I Statements are executed. After executing Block-I Statements control reaches to Statements-X.
- If the condition1 outcome is FALSE, then control reaches to condition2 and is evaluated. If condition2 is TRUE, then Block-II Statements are executed. After executing Block-II Statements control reaches to Statements-X and so on.

```
/*Example program for else-if ladder*/
#include<stdio.h>
#include<conio.h>
void main( )
{
        int c;
        clrscr( );
        printf(“enter any  character”);
        c=getchar( );
        if(c>’a’&&c<=’z’)
                printf( “The given character is lowercase character\n”);
        else if(c>=’A’&&c<=’Z’)
                printf(“The given character is uppercase character\n”);
        else if(c>=’0’&&c<=’9’)
                printf(“The given character is DIGIT\n”);
        else
                printf(“the given character is special character\n”);
        getch( );
}
Output
Enter any character: 7
The given character is DIGIT
```

```
/* C Program to find whether given year is leap year or not */
#include<stdio.h>
#include<conio.h>
void main( )
{
        int year;
        clrscr( );
        printf("Enter the Year that you want to check : ");
        scanf("%d", &year);
        if(year % 400 == 0)
                printf("%d is a Leap Year.", year);
        else if(year % 100 == 0)
                printf("%d is not a Leap Year.", year);
        else if(year % 4 == 0)
                printf("%d is a Leap Year.", year);
         else
                printf("%d is not a Leap Year", year);
          getch( );
}
OUTPUT:
Enter the Year that you want to check : 2014
2014 is not a Leap Year
```

```
/*Write a program to print electric bill paid by the customer based on
        Consumption Units                    Rate of Charge
            0 – 100                             Rs: 1.75
            101 – 200                           Rs: 2.25
            201 – 300                           Rs: 3.75
            Above 300                           Rs: 5.00             */
#include<stdio.h>
#include<conio.h>
void main( )
{
        int units;
        float rate,amount;
        clrscr( );
        printf("\nenter number of units :");
        scanf("%d",&units);
        if(units>=0&&units<=100)
                rate=1.75;
        else if(units>=101&&units<=200)
                rate=2.25;
        else if(units>=201&&units<=300)
                rate=3.75;
        else
                rate=5.00;
        amount=units*rate;
        printf("\nThe bill amount for %d units is Rs %.2f",units,amount);
        getch( );
}
```

OTUPUT:

```
enter number of units :150
The bill amount for 150 units is Rs 337.50
------------
enter number of units :320
The bill amount for 320 units is Rs 1600.00
```

```
/*Write a program to print status of the student according to the following rules:
                        Average Marks                         Grade
                             0 to 39                   FAIL
                             40 to 49                  THIRD DIVISION
                             50 to 59                  SECOND DIVISION
                             60 to 79                  FIRST DIVISION
                             80 to 100                 HONOUR           */
#include<stdio.h>
#include<conio.h>
void main( )
{
        int marks;
        clrscr( );
        printf("\nenter average marks of students :");
        scanf("%d",&marks);
        if(marks>=0&&marks<=39)
                printf("\nThe grade of student is FAIL");
        else if(marks>=40&&marks<=49)
                printf("\nThe grade of student is THIRD DIVISION");
        else if(marks>=50&&marks<=59)
                printf("\nThe grade of student is SECOND DIVISION");
        else if(marks>=60&&marks<=79)
                printf("\nThe grade of student is FIRST DIVISION");
        else if(marks>=80&&marks<=100)
                printf("\nThe grade of student is HONOUR");
        else
                printf("\nAverage is out of range");
        getch( );
}
```

OUTPUT:

```
enter average marks of students :80
The grade of student is HONOUR
---------------
enter average marks of students :62
The grade of student is FIRST DIVISION
```

Ternary operator (or) Conditional Operator:

C provides condition evaluation operator called the ternary operator y combining the following two symbols: ? :

The general format: exp1? exp1: exp2;

The ?: Operator evaluates the expression1, if it is true it returns exp1 and returns exp2 otherwise.

```
/*program to find out the max number from the given two numbers by using ternary operator* /
#include<stdio.h>
#include<conio.h>
void main( )
{
int x,y,max;
clrscr( );
printf("enter a first number");
scanf("%d",&x);
printf("enter a second number");
scanf("%d",&y);
max=x>y?x:y);
printf("max number of %d%d is :%d\n", x, y, max);
}
```

OUTPUT:

```
 Enter first number: 43
 Enter second number: 12
 Enter third number: 43
```

Switch statement:

"switch" statement works in same way as "if else-if" but it is more elegant. The switch statement is a special multi-way decision maker that tests whether an expression matches one of a number of constancy values, and branches accordingly. Switch differs from if else – if because switch can test for only equality, whether if can evaluate logical expression. The 'switch' statement is often used to process keyboard commands like menu options.

The general format:

```
switch(Expression)
{
        case value1:   Block-I Statements
                       break;
        case value2:   Block-II Statements
                       break;
        .
        .
        .
        case valuen:   Block-n Statements
                       break;
        default:       DefaultBlock Statements
}
```

Here,

- First Expression is evaluated and produces an integer value.
- Now, the expression value will be compared with case values value1, value2, ---, valuen. If any case value coincide with the expression value then that particular block statements are executed until break statement is encountered.
- break is a branch control statement used to transfer the control out of the loop.
- Case values value1, value2, …. are either integer constants (or) character constants.
- If the expression value doesn't match with any case value then default block statements will be executed.
- Default block is optional block.
- Generally switch statements are used for creating menu programs.

Note that in the above structure switch, case, break and default are C keywords.

```
/* Create a menu program to select a choice and print message as 1 for red 2 for green 3 for blue */
#include<stdio.h>
#include<conio.h>
void main( )
{
        int ch;
        clrscr( );
        printf("\n1:RED\n2:GREEN\n3:BLUE");
        printf("\nEnter Your Choice:");
        scanf("%d",&ch);
        switch(ch)
        {
                case 1:printf("\nRED SELECTED");
                        break;
                case 2:printf("\nGREEN SELECTED");
                        break;
                case 3:printf("\nBLUE SELECTED");
                        break;
                default:printf("\nINVALID SELECTION");
        }
        getch( );
}
```

OUTPUT:

```
1:RED
2:GREEN
3:BLUE
Enter Your Choice:1

RED SELECTED
```

```
/*PROGRAM TO PERFORM ARITHMETIC OPERATIONS USING MENU FORM */
#include<stdio.h>
#include<conio.h>
void main( )
{
        int x,y,ch,sum,sub,mul,div,rem;         clrscr( );
        printf("\nEnter two Numbers:");
        scanf("%d%d",&x,&y);
        while(1)
        {
                printf("\n1:ADDITION\n2:SUBTRACTION\n3:MULTIPLICATION\n");
                printf("4:DIVISION\n5:REMAINDER\n6:EXIT");
                printf("\nEnter Ur Choice:");
                scanf("%d",&ch);
                switch(ch)
                {
                        case 1: sum=x+y;
                                printf("\nADDITION RESULT:%d\n",sum);
                                break;
                        case 2: sub=x-y;
                                printf("\nSUBTRACTION RESULT:%d\n",sub);
                                break;
                        case 3: mul=x*y;
                                printf("\nMULTIPLICATION RESULT:%d\n",mul);
                                break;
                        case 4: div=x/y;
                                printf("\nDIVISION RESULT:%d\n",div);
                                break;
                        case 5: rem=x%y;
                                printf("\nREMAINDER RESULT:%d\n",rem);
                                break;
                        case 6: exit(0);
                        default: printf("\nINVALID CHOICE\n");
                }
        }
        getch( );
}
```

OUTPUT:

```
Enter two Numbers:5 3
1:ADDITION
2:SUBTRACTION
3:MULTIPLICATION
4:DIVISION
5:REMAINDER
6:EXIT
Enter Ur Choice:1
ADDITION RESULT:8
```

The switch( ) case and Nested if:

switch( ) case	Nested if
The switch( ) can only test for equality i.e. only constant values are applicable	The if can evaluate relational and logical expressions
No two case statements have identical constants in the same switch	Same conditions may be repeated for number of times
Character constants are automatically converted to integers	Character constants are automatically converted to integers
In switch( ) case statement nested if can be used	In nested if statement switch( ) case can be used

One of the most important uses of switch statement is if same block statements are necessary to executed for more than one case value, then follows the syntax as:

Syntax:
```
switch(Expression)
{
        case value1:
        case value2:
                :
                :
        case valuen: Block Statements;
                        break;
        :
        default: defaultBlockStatements
}
```

```
/* PROGRAM TO CHECK WHETHER A GIVEN CHARACTER IS VOWEL OR NOT */
#include<stdio.h>
#include<conio.h>
void main( )
{
        char ch;
        clrscr( );
        printf("\nEnter A Character:");
        scanf("%c",&ch);
        switch(ch)
        {
                case 'a':
                case 'A':
                case 'e':
                case 'E':
                case 'i':
                case 'I':
                case 'o':
                case 'O':
                case 'u':
                case 'U':   printf("\nThe entered character is VOWEL");
                            break;
                default:    printf("\nThe entered character is CONSONANT");
        }
        getch( );
}
OUTPUT:
Enter A Character:k

The entered character is CONSONANT
-------------------
Enter A Character:a

The entered character is VOWEL
```

Iterative Statements (or) Loop Constructs:

What is a loop?

A loop is defined as a block of statements which are repeatedly executed for certain number of times.

Loop variable: It is the variable used in the loop

Initialization: It is the first step in which starting and final value is assigned to the loop variable. Each time the updated value is checked by the loop itself;

Increment/Decrement: It is the numerical value added or subtracted to the variable in each round of loop.

Loop control statements are used for repetitive execution of statements depends upon the outcome of the logical test. C language provides three loop control statements as:

a) while statement

b) do-while statement

c) for statement

a)* while Loop:* The while loop in the C starts with the keyword while, followed by a parenthesized Boolean condition has a set of statements which constitute the body of the loop.

Syntax:

```
while(condition)
{
        Block Statements
}
Next Statements
```

Here,

- First the condition is evaluated. It produces either TRUE or FALSE.
- If the condition is TRUE, then Block Statements will be executed by the compiler. After executing the statements, once again control reaches to condition section. Again condition is evaluated.
- The process is repeated until the condition becomes FALSE.
- When the condition reaches to FALSE, then the control is transferred out of the loop.

EX: */*Factorial of the given number by using while loop*/*

```
#include<stdio.h>
#include<conio.h>
void main ( )
{
        int i=1,j,rst=1;
        clrscr( );
        printf("enter any integer number");
        scanf ("%d",&x);
        while (i<=x)
        {
                rst=rst*I;
                i++;
        }
        printf ("factorial of %d is: %d",x,rst);
        getch( );
}
```

OUTPUT:

```
Enter any integer number: 5
Factorial of 5 is: 120
```

b) do while loop: The do-while loop performs the test at the bottom rather than at the top. The do-while loop start with the keyboard do, followed by the body of the loop.

Syntax:

```
do
{
        -   - -
        -   - -   Block Statements
}while(condition);
Next Statements
```

Here,

- First the control executes Block statements and then enters into condition section.
- Condition is evaluated and produces either true or false.
- If the condition is true, then once again control enters into block statement and executed and soon. This procedure is repeated as long as the condition becomes true.
- If the condition reaches to false, then the control comes out of the loop and reaches to next statements available after the loop.

Note: The main difference between while and do-while statements is do-while statement executed the block statements at least once even the condition becomes false.

EX: /*Factorial of the given number by using the do-while loop*/

```
#include<stdio.h>
#include<conio.h>
void main( )
{
        int x,i=1,rst=1;
        clrscr( );
        printf (“Enter any integer number \n “);
        scanf (“%d”,&x);
        do
        {
                rst=rst*i;
                i++;
        }while(i<=x);
        printf (“factorial of %d is: %d”,x,rst);
        getch( );
}
```

c) for loop: This is used when the statements are to be executed more than once .This is the most widely used iteration construct. The for loop supported by C is much more powerful than its counterpart in other high level languages.

Syntax:

```
for(Initialization ; Condition ; Increment/Decrement)
{
        -   - -
        -   - - Block Statements
        -   - -
}
Next Statements
```

Here,

- First the control reaches to Initialization section. Initialization starts with assigning value to the variable and executes only once at the start of the loop. Then control enters into Condition section.

- In Condition section, if the outcome of the Condition is true, then control enters into Block Statements and is to be executed. After executing the statements control reaches to Increment/Decrement section.
- After incrementing or decrementing the variable in Increment/Decrement section, again control reaches to Condition section.
- This procedure is repeated as long as the condition becomes true. If the Condition becomes false, then control comes out of the loop and the control reaches to Next statements available after the loop.

```
Ex:    /*Factorial of the given number by using for loop*/
 #include<stdio.h>
#include<conio.h>
void  main ( )
{
        int x,i,rst=1;
        printf("Enter any integer number:");
        scanf("%d",&x);
        for (i=1;i<=x;i++)
                rst=rst*1;
        printf("Factorial of %d is:%d",x,rst);
}
Output:
  Enter any integer number :5
  Factorial of 5 is :120
```

Features of for loop:

1. More than one variable can be initialized at a time in the for statement. In such situations the variables are separated with comma operator. Similarly, the Increment/Decrement section may also have more than part.

```
Ex:         for(i = 1, j = n ; j >= 1 ; i++ , j--)
            {
                    printf("\n");
                    printf("%d \t %d",i,j);
            }
```

2. In condition section, it may have any compound relations.
3. An important feature of for loop is that one or more sections (Initialization and Increment/Decrement) can be omitted. In such situations, initialization has been done before the for statement and the control variable is incremented or decremented inside the loop.

NESTED LOOPS

Loop control statements can be placed within another loop control statement. Such representation is known as nested loops. In these situations, inner loop is completely embedded within outer loop.

Example:
```
for( i=1; i<=n; i++)                outer loop
{
        ----
        for( j=1; j<=i; j++)        inner loop
        {
                ---
        }
        ----
}
```

<u>Branch (or) Jump control statements:</u>

Branch control statements are used to transfer the control from one place to another place. C language provides three branch control statements. Those are

i) break statement

ii) continue statement

iii) goto statement

i) break: We already have used the break statement in switch statement. It can be also be used inside a while loop, a for loop and do-while loop. It causes control to break out the loop. Because of its nature a break will always be conditional (attached to an if).

The general format:

```
  while(1)
  {
        /*do something*/
        if(some condition)
        break;
        /*do something*/
  }
```

ii) The continue statement whenever executed causes the rest of current iteration to be skipped and causes the next iteration to begin, subjecting of course to the truth of the controlling condition.

<u>The general format:</u>

```
while(exp)
{
   /*do something*/
  if(some condition)
     continue;
   /* do something*/
}
```

iii) exit( ): exit( ) is a standard library function used to terminate the program execution.

The general format:

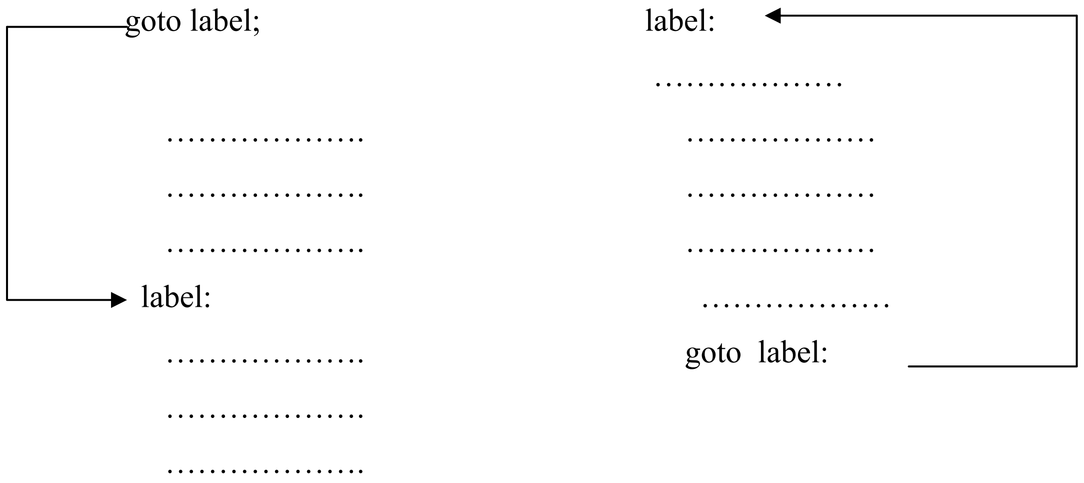
```
                    exit(argument);
```

iv) *goto* statement:

C supports the goto statement to branch unconditionally from one point to another in the program. Although it may not be essential to use the goto statement in a highly structured language like C, there may be occasions when the use of goto might be desirable.

The goto requires a label in order to identify the lace where the branch is to be made. A label is any valid variable name, and must be followed by a colon. The label is placed immediately before the statement where the control is to be transferred.

The general format:



The label can be anywhere in the program either before or after the goto label; statement. If the label: is before the statement goto label; a loop will be formed and some statements will be executed repeatedly. Such a jump is known as backward jump. On the other hand if the label: is placed after the goto label; some statements will b skipped and the jump is known n as a forward jump.

COMMA OPERATOR:

Comma operator permits two different expressions to appear in situations where only one expression would ordinarily be used. For example, it is possible to write

for (***expression 1a, expression 1b; expression 2; expression 3) ;***

where ***expression la*** and ***expression 1b*** are the two expressions, separated by the comma operator. These two expressions would typically initialize two separate indices that would be used simultaneously within the **for** loop.

Similarly, a **for** statement might make use of the comma operator in the following manner.

for (***expression* 1; *expression 2; expression 3a, expression 3b);***

Here ***expression 3a*** and ***expression 3b,*** separated by the comma operator, appear in place of the usual single expression. In this application the two separate expressions would typically be used to alter (e.g., increment or decrement) two different indices that are used simultaneously within the loop. For example, one index might count forward while the other counts backward.

Example programs:

```
/* Program to print alphabets using while */
#include <stdio.h>
#include<conio.h>
void main( )
{
          char ch = 'A';                    /* Initialization */
          clrscr( );
          while (ch <= 'Z')                 /* Test condition */
          {
                    printf("%c\t" , ch);
                    ch++;                   /* Updation by incrementing */
          }
          getch( );
}
```

Output:

```
A    B    C    D    E    F    G    H    I    J
K    L    M    N    O    P    Q    R    S    T
U    V    W    X    Y    Z
```

```
/* Program to print reverse of a number*/
#include<stdio.h>
#include<conio.h>
void main( )
{
    int n,n1,i;
    clrscr( );
    printf("enter the number ");
    scanf("%d",&n);
    n1=n;
    while(n!=0)
    {
            i=n%10;
            rev=10*rev+i;
            n=n/10;
    }
    printf("\nReverse of %d is %d",n1,rev);
    getch( );
}
```

Output:

```
enter the number 452
reverse of 452 is:254
```

/* program to print first n natural numbers*/

```
#include<stdio.h>
#include<conio.h>
void main( )
{
        int i,n;
        clrscr( );
        printf("enter a number ");
        scanf("%d",&n);
        for (i=1; i<=n; i++)
        {
                printf("%d\t", i);
        }

        getch( );
}
```

OUTPUT:

```
enter a number 20
1    2    3    4    5    6    7    8    9    10
11   12   13   14   15   16   17   18   19   20
```

/* Program to find sum of first n natural numbers*/

```
#include<stdio.h>
#include<conio.h>
void main( )
{
        int i,n,sum=0;
        clrscr( );
        printf("enter a number ");
        scanf("%d",&n);
        for (i=1; i<=n; i++)
        {
                sum = sum + i;
        }
        printf("\nSum = %d", sum);
        getch( );
}
```

Output: enter a number 15

```
Sum = 120
```

/* Program to print number from digit to words*/
```
#include<stdio.h>
#include<conio.h>
#include<math.h>
void main( )
{
        long int num,q,num1,n=0, x;
        clrscr( );
        printf("enter any number");
        scanf("%ld",&num);
        num1=num;
        while(num1!=0)
        {
                n++;
                num1=num1/10;
         }
        while(num!=0)
        {
                x=pow(10,(n-1));
                q=num/x;
                num=num%x;
                n--;
                switch(q)
                {
                         case 1:printf("\t ONE ");
                                 break;
                        case 2: printf("\t TWO ");
                                break;
                        case 3: printf("\t THREE ");
                                break;
                        case 4: printf("\t FOUR ");
                                break;
                        case 5: printf("\t FIVE ");
                                break;
                        case 6: printf("\t SIX ");
                                break;
                        case 7: printf("\t SEVEN ");
                                break;
                        case 8: printf("\t EIGHT ");
                                break;
                        case 9: printf("\t NINE ");
                                break;

                }
        }
        getch( );
}
```

Output:
```
enter any number 48
FOUR    EIGHT
```

/* Program to use nested for loop to print multiplication tables*/

```
#include <stdio.h>
#include<conio.h>
void main( )
{
        int i , n , j;
        clrscr( );
        printf("enter a number upto which to print tables");
        scanf("%d",&n);
        for (i=1 ; i<=n ; i++)
        {
                for (j=1 ; j<= 10; j++)
                {
                        printf("\n%d*%d=%d",i,j,i*j);
                }
                printf("\n--------------\n");
        }
        getch( );
}
```

Output:

```
enter a number upto which to print tables   3
1*1=1
1*2=2
1*3=3
1*4=4
1*5=5
1*6=6
1*7=7
1*8=8
1*9=9
1*10=10
--------------
2*1=2
2*2=4
2*3=6
2*4=8
2*5=10
2*6=12
2*7=14
2*8=16
2*9=18
2*10=20
--------------
3*1=3
3*2=6
3*3=9
3*4=12
3*5=15
3*6=18
3*7=21
3*8=24
3*9=27
3*10=30
```

```
/* Program to check whether the given number is prime or not */
#include<stdio.h>
#include<conio.h>
void main( )
{
    int i,n,check;
    clrscr( );
    printf("enter a number to check it is prime or not");
    scanf("%d",&n);
    check=0;
    for(i=2;i<n;i++)
    {
            if(n%i==0)
            {
                printf("%d is not a Prime number",n);
                check++;
                break;
            }
            i++;
    }
    if(check==0)
        printf("%d is Prime number",n);

    getch( );
}
```

Output:

```
enter a number to check it is prime or not153
153 is Prime number
```

/* Program to find perfect squares from 1 to n */

```
#include<stdio.h>
#include<math.h>
#include<conio.h>
void main( )
{
    int i,x,n,count;
    float c;
    clrscr( );
    printf("enter range upto which to find perfect squares");
    scanf("%d",n);
    printf("\n perfect squares from 1 to n \n");
    count=0;
    for(i=1;i<=n;i++)
    {
        c=sqrt(i);
        x=(int)c;
        if(x==c)
        {
            printf("\t %5d",i);
            count++;
        }
    }

    printf("\n total number of perfect squares = %d\n",count);

    getch( );

}
```

Output:

```
enter range upto which to find perfect squares  100
 perfect squares from 1 to n
         1      4      9     16     25     36     49     64     81    100
total number of perfect squares = 10
```

/* program to print Armstrong numbers*/

```
/*If sum of cubes of each digits of the number is equal to number itself,
then the number is called as an armstrong number (eg; 153=1^3+5^3+3^3)*/
#include<stdio.h>
#include<conio.h>
#include<math.h>
void main( )
{
    int n,x,k,i,cube,low,up,count=0;
    clrscr( );
    printf("enter range to find armstrong numbers ");
    scanf("%d%d",&low,&up);
    printf("\n The following numbers are armstrong numbers\n");
    for(k=low;k<=up;k++)
    {
        cube=0;
        n=k;
        while(n!=0)
        {
            i=n%10;
            cube=cube+pow(i,3);
            n=n/10;
        }
        if(cube==k)
        {
            printf("\t%d",k);
            count++;
        }
     }
     if(count==0)
      printf("\nthere are no armstrong numbers in the given range %d and %d",low,up);
     else
      printf("\nthere are %d armstrong numbers in the given range %d and %d",count,low,up);
    getch( );
}
```

Output:

```
enter range to find armstrong numbers(low & high): 1   500
 The following numbers are armstrong numbers
        1       153     370     371     407
there are 5 armstrong numbers in the given range 1 and 500
```

/* This program demonstrate the use of nested for loop to display Tables*/

```
#include <stdio.h>
#include<conio.h>
void main( )
{       int i , n , j;
        printf("enter a number upto which to print tables");
        scanf("%d",&n);
        printf("%5c",'*');
        for (j=1 ;j<= 10; j++)
        {
                 printf("%5d",j);
         }
         printf("\n");
         for (i=1 ;i<=n ; i++)
        {
                printf("%5d",i);
                for (j=1 ; j<= 10; j++)
                {
                        printf("%5d",i*j);
                }
                printf("\n");
        }
        getch( );
}
```

OUTPUT;

```
enter a number upto which to print tables10
   *   1    2    3    4    5    6    7    8    9    10
   1   1    2    3    4    5    6    7    8    9    10
   2   2    4    6    8   10   12   14   16   18    20
   3   3    6    9   12   15   18   21   24   27    30
   4   4    8   12   16   20   24   28   32   36    40
   5   5   10   15   20   25   30   35   40   45    50
   6   6   12   18   24   30   36   42   48   54    60
   7   7   14   21   28   35   42   49   56   63    70
   8   8   16   24   32   40   48   56   64   72    80
   9   9   18   27   36   45   54   63   72   81    90
  10  10   20   30   40   50   60   70   80   90   100
```

```
/* program to print stars as shown below
*
**
***
****
*****                          */
#include<stdio.h>
#include<conio.h>
void main( )
{
    int lines,row,column;
    clrscr( );
    printf("In how many lines stars should be printed? ");
    scanf("%d",&lines);
    for(row=1;row<=lines;row++)
    {
        for(column=1;column<=row;column++)
        {
            printf(" * ");
        }
        printf("\n");
    }
    getch( );
}
```

Output:

```
In how many lines stars should be printed? 5
 *
 *  *
 *  *  *
 *  *  *  *
 *  *  *  *  *
```

```
/* program to print numbers as shown below
1
1 2
1 2 3
1 2 3 4
4 3 2 1
3 2 1
2 1
1                                                  */
#include<stdio.h>
#include<conio.h>
void main( )
{
    int lines,row,column;
    clrscr( );
    printf("how many lines numbers should be printed?");
    scanf("%d",&lines);
    for(row=1;row<=lines;row++)
    {
        for(column=1;column<=row;column++)
        {
            printf("%3d",column);
        }
        printf("\n");
    }
    printf("\n");
    for(row=lines;row>=1;row--)
    {
        for(column=row;column>=1;column--)
        {
            printf("%3d",column);
        }
        printf("\n");
    }
    getch( );
}
```

```
Output:
how many lines numbers should be printed?5
 1
 1  2
 1  2  3
 1  2  3  4
 1  2  3  4  5
 5  4  3  2  1
 4  3  2  1
 3  2  1
 2  1
 1
```

```
/* program to print numbers as shown below
1
2 1
3 2 1
4 3 2 1
4 3 2 1
3 2 1
2 1
1                              */

#include<stdio.h>
#include<conio.h>
void main( )
{
    int lines,row,column;
    clrscr( );
    printf("how many lines numbers should be printed?");
    scanf("%d",&lines);
    for(row=1;row<=lines;row++)
    {
        for(column=row;column>=1;column--)
        {
            printf("%3d",column);
        }
        printf("\n");
    }
    printf("\n");
    for(row=lines;row>=1;row--)
    {
        for(column=row;column>=1;column--)
        {
            printf("%3d",column);
        }
        printf("\n");
    }
    getch();
}
```

Output:

```
how many lines numbers should be printed?5
  1
  2  1
  3  2  1
  4  3  2  1
  5  4  3  2  1
  5  4  3  2  1
  4  3  2  1
  3  2  1
  2  1
  1
```

```
/* program to print numbers as
1
02
003
0004
00005
0004
003
02
1                                    */

#include<stdio.h>
#include<conio.h>
void main( )
{
          int i,num;
         clrscr( );
         printf("enter any number:");
         scanf("%d",&num);
         for(i=1;i<num;i++)
                  printf("%0*d\n",i,i);
         for(i;i>=1;i--)
                  printf("%0*d\n",i,i);
         getch( );
}
```

Output:

```
enter any number:5
1
02
003
0004
00005
0004
003
02
1
```

```
/* program to convert a line of text in lower case to upper case*/
#include<stdio.h>
#include<conio.h>
#include<ctype.h>
#define EOL '\n'
void main( )
{
        char ch,text[80];
        int tag,count=0;
        clrscr( );
        /* read in the lowercase text */
        printf("enter the text in lower case\n");
        while ((ch=getchar( ))!=EOL)
        {
                text[count]=ch;
                ++count;
        }
        text[count]='\0';
        tag =count;
        printf("The text you entered is:\n");
        puts(text);
        /* display t h e uppercase text */
        printf("The text in upper case is:\n");
        count=0;
        while (count < tag)
        {
                putchar(toupper(text[count]));
                ++count;
        }
        getch( );
}
```

OUTPUT:

```
enter the text in lower case
hello this is knreddy c notes
The text you entered is:
hello this is knreddy c notes
The text in upper case is:
HELLO THIS IS KNREDDY C NOTES
```

```
/* program to print as follows
        ==========
        *          *
        *          *
        *          *
        ==========              */
#include<stdio.h>
#include<conio.h>
void main( )
{
        int rows,width,i,j;
        clrscr( );
        printf("Enter number of rows : ");
        scanf("%d",&rows);
        printf("Enter width(number of characters per line) :");
        scanf("%d",&width);
        for(i=1;i<=rows;i++)
        {
                if(i==1||i==rows)
                {
                        for(j=1;j<=width;j++)
                                printf("=");
                        printf("\n");
                }
                else
                {
                        for(j=1;j<=width;j++)
                        {
                                if(j==1||j==width)
                                        printf("*");
                                else
                                        printf(" ");
                        }
                        printf("\n");
                }
        }
        getch( );
}
OUTPUT:
Enter number of rows : 5
Enter width(number of characters per line) :10
==========
*        *
*        *
*        *
==========
```

FUNCTIONS

'C' programs are compound of a set of functions. The functions are normally used to divide a large program into smaller programs, which are easier to handle.

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

Functions serve two purposes.

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

C functions can be classified into two categories namely **LIBRARY FUNCTIONS** and **USER DEFINED FUNCTIONS**

- 'main' is an example of user defined function.
- printf, scanf, sqrt etc. belongs to the category of library functions.

The main difference between these categories is that library functions are not required to be written by us where as a user defined functions has to be developed by the user at the time of writing a program. However a user defined function can later becomes a part of the "C" program library.

- Each function normally performs a specific task.
- Every program must contain one function named as *main* where the program always begins execution. The main program may call other functions within it may call still other functions.
- If a program contains only one function, it must be **main( )**.
- If a C program contains more than one function, then one (and only one) of these functions must be **main( )**, because program execution always begins with **main( )**.
- There is no limit on the number of functions that might be present in a C program.
- Each function in a program is called in the sequence specified by the function calls in **main( )**.
- After each function has done its thing, control returns to **main( )**.When **main( )** runs out of function calls, the program ends.
- If a program contains multiple functions, their definitions may appear in any order and they must be independent of one another i.e., one function definition can't be embedded within another.

LIBRARY FUNCTIONS:

A primary goal of software engineering is to write error-free code. Code reuse, reusing program fragments that have already been written and tested whenever possible, is one way to accomplish this goal.

Library functions are pre-defined functions. A user can't understand the internal working of the function and can only use those functions. These functions can't be modified by the programmer.

'C' provides rich set of standard library functions whose definitions have been written and are ready to be used in the programs. The user must include the prototype declaration to use the standard library functions. The function prototypes for these functions are available in the general header files. Therefore, the appropriate header file for a standard function must be included at the beginning of our programs.

HEADER FILES: The entire 'C' library is divided into several files. Each such file is known as header file. Usually header files have an extension like **.h** to distinguish that from the 'C' program files.

For all the library functions, definitions and necessary information is available in header files.

Some of the important header files are:

stdio.h	-	Standard Input and Output header file
conio.h	-	Console Input and Output header file
stdlib.h	-	Standard library header file
ctype.h	-	Character type header file
math.h	-	Mathematical header file

ctype.h - Character type header file

This header file provides several library functions used to character testing and conversions.

Functions:

1. isalnum( ): Syntax: isalnum(ch)

Where, ch is a character type variable.

Function determines whether the given argument is alpha numeric or not. If the character is alpha numeric, it returns a non-zero value that represents TRUE; otherwise, it returns '0' that represents FALSE.

2. isalpha( ): Syntax: isalpha(ch)

Function determines whether the given argument is alphabet or not. If the character is alphabet, it returns a non-zero value that represents TRUE; otherwise, it returns '0' that represents FALSE.

3. isdigit( ): Syntax: isdigit(ch)

Function determines whether the given argument is digit or not. If the character is digit, it returns a non-zero value that represents TRUE; otherwise, it returns '0' that represents FALSE.

4. isspace( ): Syntax: isspace(ch)

Function determines whether the given argument is space or not. If the character is space, it returns a non-zero value that represents TRUE; otherwise, it returns '0' that represents FALSE.

5. islower( ): Syntax: islower(ch)

Function determines whether the given argument is in lowercase or not. If it is in lowercase, then returns a non-zero value that represents TRUE; otherwise, it returns '0' that represents FALSE.

6. isupper( ): Syntax: isupper(ch)

Function determines whether the given argument is in uppercase or not. If it is in uppercase, then returns a non-zero value that represents TRUE; otherwise, it returns ‘0’ that represents FALSE.

7. tolower( ): Syntax: tolower(ch)

Function converts the given argument from upper case to lower case.

8. toupper( ): Syntax: toupper(ch)

Function converts the given argument from lower case to upper case.

```
/* PROGRAM TO COUNT NUMBER OF ALPHABETS, DIGITS, WORDS AND SPECIAL
SYMBOLS IN A GIVEN LINE */
#include<stdio.h>
#include<conio.h>
#include<ctype.h>
void main( )
{
        char ch;
        int a=0,d=0,w=1,s=0;
        clrscr( );
        printf("\nEnter a Line:");
        ch=getchar( );
        while(ch!='\n')
        {
                if(isalpha(ch))
                        a=a+1;
                else if(isdigit(ch))
                        d=d+1;
                else if(isspace(ch))
                        w=w+1;
                else
                        s=s+1;
                ch=getchar( );
        }
        printf("\nALPHABETS:%d",a);
        printf("\nDIGITS:%d",d);
        printf("\nWORDS:%d",w);
        printf("\nSPECIAL SYMBOLS:%d",s);
        getch( );
}
```

OUTPUT:

```
Enter a Line:knreddy "ap21aq9996"&"aaq6305"
ALPHABETS:14
DIGITS:10
WORDS:2
SPECIAL SYMBOLS:5
```

math.h - Mathematical header file

This header file provides several library functions used for mathematical operations.

1. sin( ): Syntax: sin(d)

Where, d is a double type argument.

Function receives a double type argument and returns the sine value as double.

Ex: printf("%lf", sin(45));

2. cos( ): Syntax: cos(d)

Function receives a double type argument and returns the cosine value as double.

Ex: printf("%lf", cos(45));

3. tan( ): Syntax: tan(d)

Function receives a double type argument and returns the tangent value as double.

Ex: printf("%lf", tan(45));

4. log( ): Syntax: log(d)

Function receives a double type argument and returns the natural logarithm (base e) value as double.

Ex: printf("%lf", log(10));

5. log10( ): Syntax: log10(d)

Function receives a double type argument and returns the logarithm (base 10) value as double.

Ex: printf("%lf", log10(10));

6. sqrt( ): Syntax: sqrt(d)

Function receives a double type argument and returns the square root value as double.

Ex: printf("%lf", sqrt(81));

7. pow( ): Syntax: pow(d1, d2)

Function receives two double type arguments and returns the d1 raised to the d2 power value as double.

Ex: printf("%lf", pow(3,4));

8. exp( ): Syntax: exp(d)

Function receives a double type argument and returns the e to the power d value as double.

Ex: printf("%lf", exp(4));

9. ceil( ): Syntax: ceil(d)

Function receives a double type argument and returns the value rounded up to the next lowest integer as double.

Ex: printf("%lf", ceil(3.85));

10. floor( ): Syntax: floor(d)

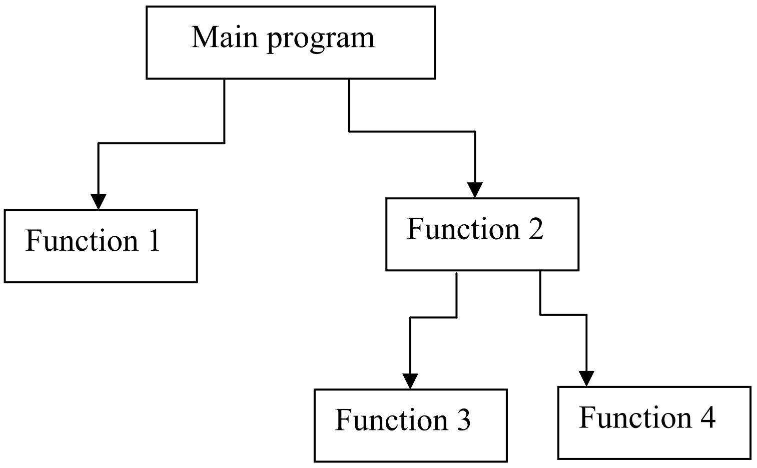
Function receives a double type argument and returns the value rounded down to the previous highest integer as double.

Ex: printf("%lf", floor(3.85));

USER DEFINED FUNCTIONS

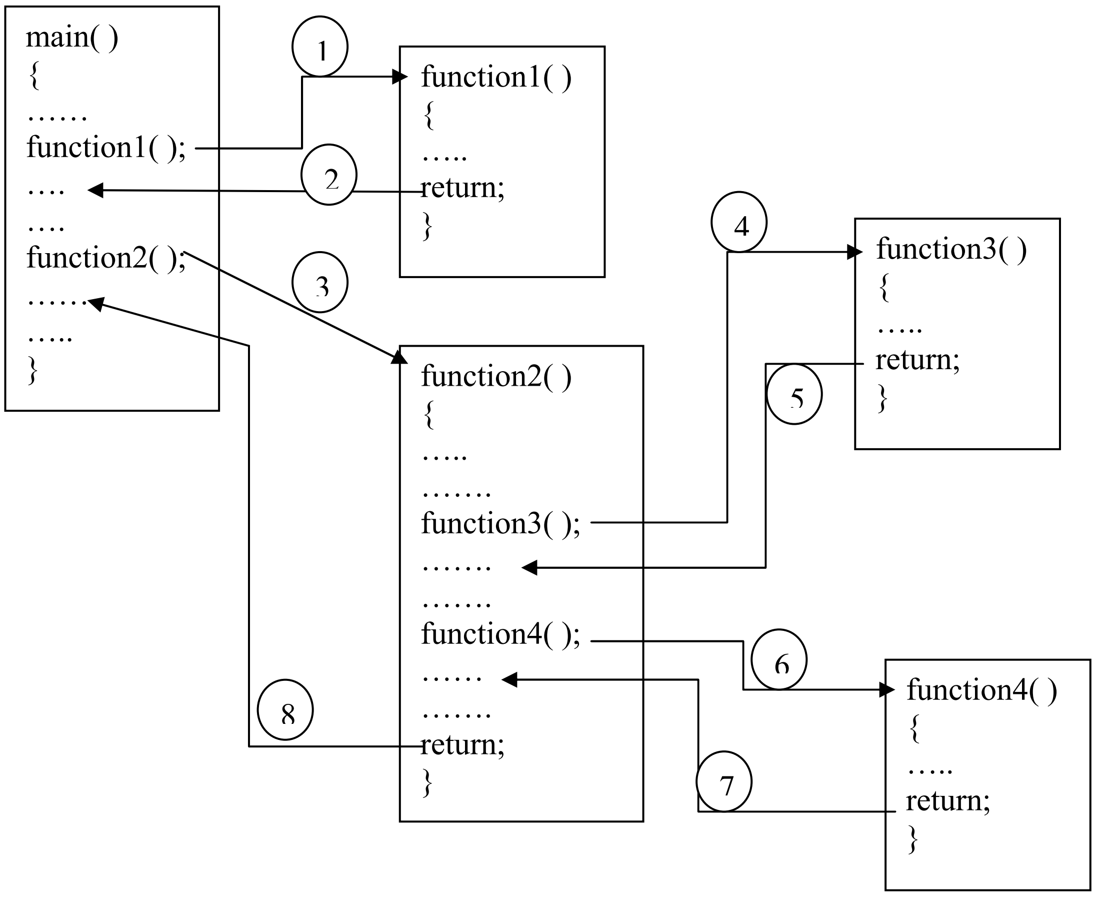
A function is a module or block of program code which deals with a particular task. Making functions is a way of isolating one block of code from other independent blocks of code.

Top down modular programming using functions:



When a function is called, program execution is transferred to the first statement on the called function. The function is completed when it executes the last statement in the function (or) it executes a return statement. Execution continues with the evaluation of the expression in which the call was made. A value can be written when a function completes and that return value can be used as an operand on an expression.

Execution of a program made of several functions



Advantages of using functions:
1. To facilitates top-down modular programming. In this programming style, the high level logic of the overall problem is solved first while the details of the each lower level function or addressed later. (i.e**., procedural abstraction**)
2. The length of the source program is reduced by using functions at appropriate places. This factor is particularly critical with microcomputers where memory space is limited.
3. It is easy to locate and isolate a faulty function for further investigations.
4. A function may be used by many other programs. This means that a C programmer can build on what other have already done, instead of starting over, from scratch.
5. Another advantage of sin function subprograms is that functions can be executed more than once in a program (i.e. **Code Reuse**).

<u>ELEMENTS OF USER-DEFINED FUNCTIONS</u>:

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 /prototype

1<u>. FUNCTION DEFINITION:</u> The actual code of a function is known as function definition.

When you create a function, you need to specify the **function header** as the first line of the function definition, followed by the executable code for the function enclosed between braces. The block of code between braces following the function header is called the **function body**.

• The function header defines the name of the function, the function parameters (in other words, what types of values are passed to the function when it's called), and the type for the value that the function returns.

• The function body determines what calculations the function performs on the values that are passed to it.

The general form of a function is essentially the same as you've been using for main(), and it looks like this:

<u>General form</u>:

```
                                                              FUNCTION HEADER
    return_type   function_name(parameters (or) arguments declaration)   ↗
    {
        local variable declaration;  ┐
        statement 1;                 │
        statement 2;                 │
        statement 3;                 │
          .                          ├  FUNCTION BODY
          .                          │
        statement n;                 │
        return(expression);          ┘
    }
```

The return type specifies the type of value that the function is expected to return to the program calling the function. If the function is used in an expression or as the right side of an assignment statement, the return value supplied by the function will effectively be substituted for the function in its position. The type of value to be returned by a function can be specified as any of the legal types in C, including pointers. The type can also be specified as void, meaning that no value is returned. A function with a void return type can't be used in an expression or anywhere in an assignment statement.

The **function name** is any valid C identifier and therefore must follow the same rules of formation as other variable names in C. The name of a function can be any legal name in C that isn't a reserved word (such as int, double, sizeof, and so on) and isn't the same as the name of another function in your program. The name should be appropriate to the task performed by the function.

Function parameters are defined within the function header and are placeholders for the arguments that need to specify the type of value that should be passed to when the function is called. The parameters for a function are a list of parameter names and their types, and successive parameters are separated by commas. A parameter is used within the body of the function to refer to that value when the function executes. The term **argument** refers to the value that you supply corresponding to a parameter when you call a function.
Parameters provide the means by which you pass information *from* the calling function *into* the function that is called. These parameter names are local to the function, and the values that are assigned to them when the function is called are referred to as arguments. The computation in the body of the function is then written using these parameter names, which will have the values of the arguments when the function executes. Of course, a function may also have locally defined automatic variables declared within the body of the function.

If the type of an argument to a function does not match the type of the corresponding parameter, the compiler will insert a conversion of the argument value to the parameter type where this is possible. This may result in truncation of the argument value, when you pass a value of type double for a parameter of type int, so this is a dangerous practice. If the compiler cannot convert an argument to the required type, you will get an error message.

The **function body** contains local declarations, statements for performing particular task and a return statement that returns a value evaluated by the function. The statements in the function body can be absent, but the braces must be present. If there are no statements in the body of a function, the return type must be void, and the function will have no effect. The type void means "absence of any type," so here it means that the function doesn't return a value. A function that does not return a value must also have the return type specified as void. Conversely, for a function that does not have a void return type, every return statement in the function body must return a value of the specified return type.

■**Note** The **statements** in the body of a function can also contain nested blocks of statements. But you can't define a function inside the body of another function.

A local variable is a variable that is defined inside a function and used without having any role in the communication between functions.

The return Statement provides the means of exiting from a function and resuming execution of the calling function at the point from which the call occurred. In its simplest form, the return statement is just this:

return;

In this form, the return statement is being used in a function where the return type has been declared as void. It doesn't return any value. However, the more general form of the return statement is this:

return expression;

This form of return statement must be used when the return value type for the function has been declared as some type other than void. The value that's returned to the calling program is the value that results when expression is evaluated.

■***Caution*** *You'll get an error message if you compile a program that contains a function defined with a void return type that tries to return a value. You'll get an error message from the compiler if you use a bare return in a function where the return type was specified to be other than void.*

The return expression can be any expression, but it should result in a value that's the same type as that declared for the return value in the function header. If it isn't of the same type, the compiler will insert a conversion from the type of the return expression to the one required where this is possible. The compiler will produce an error message if the conversion isn't possible. There can be more than one return statement in a function, but each return statement must supply a value that is convertible to the type specified in the function header for the return value.

2. **FUNCTION CALL:** The statement which is used to call a function is called a function call. You call a function by using the function name followed by the arguments to the function between parentheses. When you actually call the function by referencing it in some part of your program, the arguments that you specify in the call will replace the parameters in the function. As a result, when the function executes, the computation will proceed using the values that you supplied as arguments. The arguments that you specify when you call a function need to agree in type, number, and sequence with the parameters that are specified in the function header. If the type of an argument to a function does not match the type of the corresponding parameter, the compiler will

insert a conversion of the argument value to the parameter type where this is possible. This may result in truncation of the argument value, when you pass a value of type double for a parameter of type int, so this is a dangerous practice. If the compiler cannot convert an argument to the required type, you will get an error message. If there are no parameters to a function, you can specify the parameter list as void, as you have been doing in the case of the main( ) function.

The general form for calling a function is the following expression:

Function_name(List of Arguments - separated by commas)

<u>Parameters (or) Arguments</u>: A calling function can send a package of values for the called function to operate on it. Each value passed in this manner is called as parameter or argument. (**Parameters are nothing but input information given to a function).**

Depending on the location of arguments in a program, they are categorized into two types.

1. Actual arguments 2. Formal arguments

1. Actual arguments: The parameters which are passed in a function call are known as actual parameters. They are defined in calling function and they can be expressions or constants and variables.

<u>Note</u>: The variables used as actual arguments must be assigned values before function call is made.

2. **Formal arguments:** The parameters which are used in function definition are called formal parameters. Formal parameters can only be variables. The value of actual parameter is passed to the appropriate formal parameter.

<u>Correspondence between Actual parameters and Formal parameters</u>:

- Arguments, which are specified in function call, are called actual parameters.
- The value of an actual parameter is passed to the formal parameter.

  The actual and formal parameters should match in **number, type and order (or) sequence**

  The value of an actual argument is passed to the formal arguments on a **one-to-one** basis, starting with the first argument.

Arguments matching process between function call and called function:

```
main( )
{
        Fun1(a1, a2, ….., aM);
}
Fun1(b1, b2, …., bN)
{
        …….
}
```

- ➢ If **M >N**, the extra actual arguments are discarded.
- ➢ If **M < N**, the unmatched formal parameters are initialized to some garbage value.

Note:

- Any mismatch in data type may also result in passing of garbage value.
- No error message will be generated.
- The variables used in the actual parameters must be assigned before the function call is made.
- When a function call is made, only a copy of the values of actual arguments is passed into the called function. Whatever occurs inside the function will have no effect on the variables used in the actual argument list. (i.e., Call by Value)

3. <u>FUNCTION PROTOTYPE/ FUNCTION DECLARATION</u>: It specifies the complete information about the function. Whenever we are calling function before defining that function, we need to use prototype for that function before the **main** function. The prototype for a function is identical to the function header, with a **semicolon** added at the end. Like the function header, the function prototype includes information about the **function's return type, name, and parameters**. The prototype's job is to tell the compiler about the function's return type, name, and parameters. With this information, the compiler can check every time your source code calls the function and verify that you're passing the correct number and type of arguments to the function and using the return value correctly. If there's a mismatch, the compiler generates an error message.

Note:

- The parameter list must be separated by commas.
- The parameter names do not need to be the same in the prototype declaration and the function definition
- The types must match the types of parameters in the function definition, in number and order.
- Use of parameter names in the declaration is optional.
- If the function has no formal parameters, the list is written as void.
- The return type is optional, when the function returns **int** type data.
- The return type must be void if no value is returned.
- When the declared type does not match with the types in the function definition, compiler will produce an error.

Boolean functions:

Boolean values are represented by integers in C. It is very easy to write a function that returns either **true (or) false** value. Such a function is useful when a particular test is used many times especially when the test is somewhat complex. It can save considerable programming effort.

Ex: /*Program to find the product of two numbers*/

```
#include<stdio.h>
#include<conio.h>
int  Mul(int a,int b);       /*function prototype*/
void main( )
{
        int  a=10,b=5,m;
        clrscr( );
        m=Mul(a,b);          /*function call*/
        printf("%d",m);
        getch( );
 }
int Mul(int x,int y)
{
        int res;                /*function definition*/
        res=x*y;
        return(res);
}
OUTPUT: 50
```

Ex: /* Write a C program to find whether the given 'year' is leap or not.*/

```
#include<stdio.h>
#include<conio.h>
int is_leap(int year);
void main( )
{
        int year;
        clrscr( );
        printf("Enter year");
        scanf("%d",&year);
        if(is_leap(year))
                printf("%d is a leap year",year);
        else
                printf("%d is not a leap year",year);
        getch( );
}
int is_leap(int year)
{
        return( year%4==0&&year%100!=0||year%400==0);
}
OUTPUT:
Enter year2012
2012 is a leap year
Enter year2014
2014 is not a leap year
```

FUNCTION CATEGORIES

Depending on whether arguments are passed or not and function returns any value or not, functions are classified into different categories as:

Case 1:	Functions with no arguments and no return value
Case 2:	Functions with arguments and no return value
Case 3:	Functions with arguments and with return value

Case 1: When a function has no arguments, it does not receive any data from the calling function. Similarly when it does not return any value, the calling function does not receive any data from the called function.

Note: If the function does not return any value, return type must be **void.**

Case 2: In this case, calling function sends data to the called function. But, called function does not return any value after completing function implementation.

Case 3: In this case, calling function sends data to the called function and called function also returns some value to the function call after completing the function implementation.

PASS BY VALUE AND PASS BY REFERENCE/ CALL BY VALUE AND CALL BY REFERENCE/COMMUNICATION AMONG FUNCTIONS.

There are two ways that a computer language can pass an argument to a function:

- **Call by value**
- **Call by reference**

Call by value: A copy of the argument's value is made and passed to the called function. Therefore, changes to the copy do not affect the original variable's values in the caller function. The function works with a copy of the arguments sent to it.

Call by reference: The caller gives the called functions the ability to directly access the caller's data. The function receives reference to variable and works directly with the original variable.

The use of pointer variables as actual parameters for communicating data between functions is called **"pass by value"** or **"call by address or reference"**

Rules for pass by pointers:

- The type of the actual and formal arguments must be same
- The actual arguments (in the function call) must be the addresses of variables that are local to the calling function
- The formal arguments in the function header must be prefixed by the indirection operator *.
- In the prototype, the arguments must be prefixed by the symbol *.
- To access the value of an actual argument in the called function, we must use the corresponding formal arguments prefixed with the indirection operator *.

```
/*  CALL BY VALUE*/
#include<stdio.h>
#include<conio.h>
void swap(int x ,int y);
void main( )
{
        int a,b;
        clrscr( );
        printf("enter any value for a:");
        scanf("%d",&a);
        printf("enter any value for b:");
        scanf("%d",&b);
        printf("\nvalues of a and b before function call a=%d\tb=%d",a,b);
        swap(a,b);
        printf("\nvalues of a and b after function call a=%d\tb=%d",a,b);
        getch( );
}
void swap(int x, int y)
{
        int temp;
        printf("\nvalues of x and y before swapping x=%d\ty=%d",x,y);
        temp=x;
        x=y;
        y=temp;
        printf("\nvalues of x and y after swapping x=%d\ty=%d",x,y);
  }
```

OUTPUT:

```
enter any value for a:10
enter any value for b:20
values of a and b before function call a=10      b=20
values of x and y before swapping x=10  y=20
values of x and y after swapping x=20   y=10
values of a and b after function call a=10      b=20
```

```
/*CALL BY REFERENCE*/
#include<stdio.h>
#include<conio.h>
void swap(int *x ,int *y);
void main( )
{
        int a,b;
        printf("enter any value for a:");
        scanf("%d",&a);
        printf("enter any value for b:");
        scanf("%d",&b);
        printf("\nvalues of a and b before function call a=%d\tb=%d",a,b);
        swap(&a,&b);
        printf("\nvalues of a and b after function call a=%d\tb=%d",a,b);
        getch( );
}
void swap(int *x, int *y)
{
        int temp;
        printf("\nvalues of x and y before swapping x=%u\ty=%u",x,y);
        printf("\nvalues of *x and *y before swapping x=%d\ty=%d",*x,*y);
        temp=*x;
        *x=*y;
        *y=temp;
        printf("\nvalues of x and y after swapping x=%u\ty=%u",x,y);
        printf("\nvalues of *x and *y after swapping x=%d\ty=%d",*x,*y);
}
```

OUTPUT:

```
enter any value for a:10
enter any value for b:20
values of a and b before function call a=10     b=20
values of x and y before swapping x=2293572     y=2293568
values of *x and *y before swapping x=10        y=20
values of x and y after swapping x=2293572      y=2293568
values of *x and *y after swapping x=20 y=10
values of a and b after function call a=20      b=10
```

Call by value should be used whenever the called function does not need to modify the value of the caller's original variables value. This prevents the accidental side effects that so greatly hinder the development of correct and reliable software systems. Call by reference should only be used with trusted called functions that need modify the original variable. In C, all calls are call by value. But, it is possible to simulate call by reference using pointers

SCOPE OF VARIABLES:

Scope relates to any object that is declared, such as variable declaration and function declaration. It determines the part of the program where the object is visible. The scope of local variables is limited to the functions in which they are declared, or in other words these variables are inaccessible outside of the function .Likewise the scope of the block variables is limited to the block in which they are declared. Global have a scope that spans the entire source program, which is why they can be used in any function.

SCOPE: The region of a program in which a variable is available for use

VISIBILITY: The program's ability to access a variable from the memory

LIFETIME: The lifetime of a variable is the duration of time in which a variable exists in the memory during execution.

Classification of the variables based on the place of declaration:

i) Local variables: A variable declared inside a function called a local variable. This name derives from the fact that a variable declared inside a function can be used only inside that function.

ii) Global variables: The variables you declare in the global variable section are called global variables or external variables. While the local variable can be used inside the function in which it is declared. A global variable can be used anywhere in the program.

GLOBAL VARIABLES VS LOCAL VARIABLES:

1. Local variables can be used only inside the function of the block in which they are declared. On the other hand global variables are used throughout the program.
2. **All global variables, in the absence of explicit initialization, are automatically initialized to zero**. A global *int* variables starts up with the value 0, a global *float* gets initialized to 0.0, a global *char* holds the ASCII null byte and the global pointer points to NULL. Local variable do not get initialized to any specific value when you do not provide any value. Thus a local variable starts up with an unknown value, which may be different each time.
3. Global variables get initialized only once, typically just before the program starts executing. But local variables get initialized each time the function or block containing their declaration is entered.

4. The initial value that you supplied for a global variable must be a constant, where as a local variable can contain variable in its initialization process.
5. A local variable loses its value the moment the function/block containing it is exited. So you cannot expect a local variable to retain the value deposited in it the previous time the function/block was entered. Global variables retain their values through the program's execution.

THE CONCEPT OF GLOBAL VARIABLES

Introduction: The various modules can share information by using *global variables*.

Program

```
#include <stdio.h>
#include<conio.h>
int i =0;     //Global variable
void f1(void) ;
void main( )
{
   int j;          // local variable in main
   f1( ) ;
   i =0;
   printf("value of i in main %d\n",i);
   f1( );
   printf("value of i after call%d\n",i);
   getch( );
}
```

```
void f1(void)
{
   int k;           // local variable for f1.
   i = 50;
}
```

```
OUTPUT:
value of i in main  0
value of i after call 50
```

Explanation

1. When you define a variable inside the function block it is called a local variable.
2. The local variable can be accessed only in the block in which it is declared.
3. j is the local variable for main and it can be accessed only in the block main. That means you cannot access it in function f1.
4. k is the local variable for function f1 and it cannot be accessed in main.
5. The variable i, which is outside main, is called a global variable. It can be accessed from function main as well as function f1.
6. Any expression in this function is going to operate on the same i.
7. When you call function f1, which sets the value of i to 50, it is also reflected in main because main and f1 are referring to the same variable, i.

Points to Remember

1. Global variables can be accessed in all the functions in that file.
2. Any update to the global variable also affects the other functions, because all functions refer to the same value of i.
3. When you want to share information between multiple functions, you can use the concept of global variables.

EG:

```
#include <stdio.h>
#include<conio.h>
int x = 999;
void print_value(void);
void main( )
{
 clrscr( );
 printf("%d\n", x);
 print_value();
 getch( );
}
void print_value(void)
{
 printf("%d\n", x);
}
```

This program compiles and runs with no problems and output is:

OUTPUT:

```
999
999
```

EG:

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

void print_value(void);
void main( )
{
 int x = 999;
clrscr ( );
 printf("%d\n", x);
 print_value();
 getch( );
}
void print_value(void)
{
 printf("%d\n", x);
}
```

OUTPUT:

```
Error: undefined identifier `x'.
```

<u>RECURSION</u>

The C language supports recursive feature, i.e., the function, which is called by itself is called recursive function and this process often referred as **recursion.**

Ex:-

```
main( )
{
        printf("welcome to BTECH\n");
        main( );
}
```

Execution must be terminated abruptly; otherwise the execution will continue indefinitely.

<u>Important conditions</u>: There are two important conditions that must be satisfied by any recursive procedure.

1. Each time a procedure calls itself, it must be nearer to a solution.
2. There must be a decision criterion for stopping the computation. [Base Case]

<u>There are two types of recursions</u>:

1. **Direct recursion**: Here, the function calls itself till the condition is true.

Ex:

```
void sum( )
        {
                --
                sum( );
                --
        }
```

2. **Indirect recursion**: Here, a function calls another function, then that function (called function) calls the calling function.

Ex:

```
void sum( )
        {
                ---
                call( );
                ---
        }
        void call( )
        {
                --
                sum( );
        }
```

This concept can be explained by reference to the factorial function:

The factorial of a number is the product of the integral values from 1 to the number. We defined the factorial of a value like this:

$$\text{factorial}(n) = \begin{cases} 1 & \text{if } n=0 \\ \mathbf{n*(n-1)*(n-2)*\ldots\ldots\ldots*(3)*(2)*(1)} & \mathbf{if\ n>0} \end{cases}$$

For example, we can compute

4! = (4)(3)(2)(1)

Recall that we implemented a program to compute factorials using a simple loop that accumulates the factorial product.

Looking at the calculation of 5! You will notice something interesting. If we remove the 5 from the front, what remains is a calculation of 4!

In general, **n! = n(n -1)!.**

In fact, this relation gives us another way of expressing what is meant by factorial in general. Here is a recursive definition:

$$\mathbf{factorial(n)} = \begin{cases} \mathbf{1} & \mathbf{if\ n=0} \\ \mathbf{n*factorial(n-1)} & \mathbf{if\ n>0} \end{cases}$$

This definition says that the factorial of 0 is, by definition, 1, while the factorial of any other number is defined to be that number times the factorial of one less than that number.

Even though this definition is recursive, it is not circular. In fact, it provides a very simple method of calculating a factorial of a given number '**n**'.

Consider the value of 4!, by definition we have

4! = 4(4-1)! = 4(3!)

But what about **3!**? To find out, we apply the definition again.

4! = 4(3!) = 4[(3)(3-1)!] = 4(3)(2!)

Now, of course, we have to expand **2!**, which requires **1!**, which requires 0!. Since 0! is simply 1, that's the end of it.

4! = 4(3!) = 4(3)(2!) = 4(3)(2)(1!) = 4(3)(2)(1)(0!) = 4(3)(2)(1)(1) = 24

You can see that the recursive definition is not circular because each application causes us to request the factorial of a smaller number. Eventually we get down to 0, which doesn't require another application of the definition. This is called a ***base case*** for the recursion. When the recursion bottoms out, we get a closed expression that can be directly computed.

All good recursive definitions have these key characteristics:

1. There are one or more base cases for which no recursion is required.
2. All chains of recursion eventually end up at one of the base cases.

Recursive Functions

If we write factorial as a separate function, the recursive definition translates directly into code.

```
fact(n)
{
    if(n= =0)
        return 1;
   else
        return (n * fact(n-1));
}
```

Do you see how the definition that refers to itself turns into a function that calls itself? This is called a *recursive function*. The function first checks to see if we are at the base case n == 0 and, if so, return 1. If we are not yet at the base case, the function returns the result of multiplying n by the factorial of n-1. The latter is calculated by a recursive call to **fact (n-1**).

Remember that each call to a function starts that function a new. That means it has its own copy of any local values, including the values of the parameters.

Note especially how each return value is multiplied by a value of n appropriate for each function invocation. The values of n are stored on the way down the chain and then used on the way back up as the function calls return. There are many problems for which recursion can yield an elegant

Advantages:

- The code may be much easier to write.
- Some situations are naturally recursive.

Disadvantages:

- Recursive functions are generally slower than non-recursive functions.
- Excessive recursion may overflow the run-time stack.
- One must be very careful when writing recursive functions; they can be tricky.
- Sometimes each function call generates two or more recursive calls at that level. This has the potential to consume an enormous amount of time.

and efficient solution

/* Program to find factorial of a number by using recursive and non-recursive functions*/

NON-RECURSIVE

```
#include<stdio.h>
#include<conio.h>
unsigned int factorial(int n);
void main ( )
{
        int n,i;
        long fact;
        clrscr( );
        printf("Enter the number: ");
        scanf("%d",&n);
        if(n==0)
        printf("Factorial of 0 is 1\n");
        else
        {
                printf("Factorial of %d Using Non-Recursive Function is %d\n",n,factorial(n));
        }
        getch( );
}
/* Non-Recursive Function*/
unsigned int factorial(int n)
 {
        int fact = 1;
        int i;
        for(i = 1; i <= n; i++)
        {
                fact *= i;
        }
   return(fact);
}
OUTPUT:
Enter the number: 5
Factorial of 5 Using Non-Recursive Function is 120
```

RECURSIVE

```
#include<stdio.h>
#include<conio.h>
long int recr_factorial(int n);
void main( )
{
        int n,i;
        long fact;
        clrscr( );
        printf("Enter the number: ");
        scanf("%d",&n);
        if(n==0)
                printf("Factorial of 0 is 1\n");
        else
        {
                printf("Factorial of %d Using Recursive Function is %ld\n",n,recr_factorial(n));
        }
        getch( );
}
/* Recursive Function*/
long int recr_factorial(int n)
{
    int fact=1;
    if(n==0)
          return(1);
    else
          return (n * recr_factorial(n-1));
}
```

OUTPUT:

```
Enter the number: 5
Factorial of 5 Using Recursive Function is 120
```

/* Program to print Fibonacci series by using recursive and non-recursive functions*/

NON-RECURSIVE

```
#include<stdio.h>
#include<conio.h>
int fibonacci(int );
 void main( )
{
        int fib,no,i;
        clrscr( );
        printf("How many number of elements is to\nprint in the fibonacci series: ");
        scanf("%d",&no);
        printf("FIBONACCI SERIES IS:\n");
        if(no==0)
                printf("you had entered number zero ;hence no fibonacci series" );
        for(i=0;i<no;i++)
        {
                fib=fibonacci(i);
                printf("\t%d",fib);
        }
        getch( );
}
int fibonacci(int x)
{
        static int fib,a=0,b=1;
        if(x==0)
                return(0);
        else if(x==1)
                return(1);
        fib=a+b;
        a=b;
        b=fib;
        return(fib);
}
```

OUTPUT:

```
How many number of elements is to
print in the fibonacci series: 5
FIBONACCI SERIES IS:
 0      1      1      2      3
```

RECURSIVE

```
#include<stdio.h>
#include<conio.h>
int fibonacci(int );
 void main( )
{
        int fib,no,i;
        clrscr( );
        printf("How many number of elements is to\nprint in the fibonacci series: ");
        scanf("%d",&no);
        printf("FIBONACCI SERIES IS:\n");
        if(no==0)
                printf("you had entered number zero ;hence no fibonacci series" );
        for(i=0;i<no;i++)
        {
                fib=fibonacci(i);
                printf("\t%d",fib);
        }
        getch( );
}
int fibonacci(int x)
{
        if(x==0)
                return(0);
        else if(x==1)
                return(1);
        return(fibonacci(x-2)+fibonacci(x-1));
}
```

OUTPUT:

```
How many number of elements is to
print in the fibonacci series: 5
FIBONACCI SERIES IS:
        0       1       1       2       3
```

Some problems are easier to solve by recursive functions than linear (or) Non-recursive functions. One of such classical problem is **Towers of Hanoi.**

Towers of Hanoi Problem:

One very elegant application of recursive problem solving is the solution to a mathematical puzzle usually called the Tower of Hanoi or Tower of Brahma. This puzzle is generally attributed to the French mathematician **´Edouard Lucas,** who published an article about it in 1883. The legend surrounding the puzzle goes something like this:

Somewhere in a remote region of the world is a monastery of a very devout religious order. The monks have been charged with a sacred task that keeps time for the universe. At the beginning of all things, the monks were given a table that supports three vertical posts. On one of the posts was a stack of 64 concentric golden disks. The disks are of varying sizes and stacked in the shape of a beautiful pyramid. The monks were charged with the task of moving the disks from the first post to the third post. When the monks have completed their task, all things will crumble to dust and the universe will end.

To maintain divine order, the monks must abide by certain rules:

1. Only one disk may be moved at a time.
2. A disk may not be set aside. It may only be stacked on one of the three posts.
3. A larger disk may never be placed on top of a smaller one.

The task is to move the disk from the first post to the third post using the center post as sort of a temporary resting place during the process. Of course, you have to follow the three sacred rules given above.

We want to develop an algorithm for this puzzle. You can think of our algorithm either as a set of steps that the monks need to carry out, or as a program that generates a set of instructions. For example, if we label the three posts A, B, and C.

The instructions might start out like this:

Move disk from A to C.
Move disk from A to B.
Move disk from C to B.
...

The solution process is actually quite simple, if you know about recursion.

Let's start by considering some really easy cases. Suppose we have a version of the puzzle with only one disk. Moving a tower consisting of a single disk is simple enough; we just remove it from **A** and put it on **C**. Problem solved. OK.

What if there are two disks? I need to get the larger of the two disks over to post **C**, but the smaller one is sitting on top of it. I need to move the smaller disk out of the way, and I can do this by moving it to post **B**. Now the large disk on **A** is clear; I can move it to **C** and then move the smaller disk from post **B** onto post **C**.

Now let's think about a tower of size three. In order to move the largest disk to post **C**, I first have to move the two smaller disks out of the way. The two smaller disks form a tower of size two. Using the process I outlined above, I could move this tower of two onto post **B**, and that would free up the largest disk so that I can move it to post **C.** Then I just have to move the tower of two disks from post **B** onto post **C**. Solving the three disk case boils down to three steps:

1. Move a tower of two disks from A to B.
2. Move one disk from A to C.
3. Move a tower of two disks from B to C.

The first and third steps involve moving a tower of size two. Fortunately, we have already figured out how to do this. It's just like solving the puzzle with two disks, except that we move the tower from **A** to **B** using **C** as the temporary resting place, and then from **B** to **C** using **A** as the temporary resting place.

We have just developed the outline of a simple recursive algorithm for the general process of moving a tower of any size from one post to another.

Recursive solution for the Towers of Hanoi Problem:

Algorithm: move n-disk tower from source to destination via resting place

1) Move n-1 disks r from source to resting place
2) Move nth disk tower from source to destination
3) Move n-1 disks from resting place to destination

What is the base case for this recursive process?

Notice how a move of **n** disks results in two recursive moves of **'n-1'** disks. Since we are reducing **n** by one each time, the size of the tower will eventually be **1**. A tower of size 1 can be moved directly by just moving a single disk; we don't need any recursive calls to remove disks above it.

Fixing up our general algorithm to include the base case gives us a working **moveTower algorithm.** Our **moveTower** function will need parameters to represent the size of the tower, **n**; the source post, **source**; the destination post, **dest**; and the temporary resting post, **temp**.

We can use an int for **n** and character type for **A,. .B,.** and .**C**. to represent the posts. Here is the code for moveTower:

```
 moveTower(n, source, dest, temp)
{
        if (n = = 1)
                printf( "Move disk from", source, "to", dest");
        else
                moveTower(n-1, source, temp, dest);
                moveTower(1, source, dest, temp);
                moveTower(n-1, temp, dest, source);
}
```

See how easy that was?

To get things started, we just need to supply values for our four parameters. Let's write a little function that prints out instructions for moving a tower of size **n** from post **A** to post **C**.

Now we're ready to try it out. Here are solutions to the three- and four-disk puzzles. You might want to trace through these solutions to convince yourself that they work.

```
>>> hanoi(3)
Move disk from A to C.
Move disk from A to B.
Move disk from C to B.
Move disk from A to C.
Move disk from B to A.
Move disk from B to C.
Move disk from A to C.
>>> hanoi(4)
Move disk from A to B.
Move disk from A to C.
Move disk from B to C.
Move disk from A to B.
Move disk from C to A.
Move disk from C to B.
Move disk from A to B.
Move disk from A to C.
Move disk from B to C.
Move disk from B to A.
Move disk from C to A.
Move disk from B to C.
Move disk from A to B.
Move disk from A to C.
Move disk from B to C.
```

So, our solution to the Tower of Hanoi is a trivial. Algorithm requiring only nine lines of code.

Why this problem is is labeled as a *hard problem*?

To answer that question, we have to look at the efficiency of our solution. Remember, when I talk about the efficiency of an algorithm, I mean how many steps it requires to solve a given size problem. In this case, the difficulty is determined by the **number of disks in the tower.**

The question we want to answer is *how many steps does it take to move a tower of size* **n**?

Just looking at the structure of our algorithm, you can see that moving a tower of size **n** requires us to move a tower of size **n-1** twice, once to move it off the largest disk, and again to put it back on top. If we add another disk to the tower, we essentially double the number of steps required to solve it.The relationship becomes clear if you simply try out the program on increasing puzzle sizes.

Number of Disks	Steps in Solution
1	1
2	3
3	7
4	15
5	31

In general, solving a puzzle of size '**n**' will require **2^n -1** steps.

Computer scientists call this an *exponential time* algorithm, since the measure of the size of the problem**, n,** appears in the exponent of this formula.

Exponential algorithms blow up very quickly and can only be practically solved for relatively small sizes, even on the fastest computers.

Just to illustrate the point, if our monks really started with a tower of just 64 disks and moved one disk every second, 24 hours a day, every day, without making a mistake, it would still take them **over 580 *billion* years** to complete their task.

Even though the algorithm for Towers of Hanoi is easy to express, it belongs to a class known as ***intractable*** problems. These are problems that require too much computing power (either time or memory) to be solved in practice, except for the simplest cases.

Applications of Towers of Hanoi Problem:

1. The Towers of Hanoi problem is frequently used in psychological research on problem solving.
2. The Towers of Hanoi problem is also used as *backup rotation scheme* when performing computer data backups where multiple apes/media are involved.

3. The Towers of Hanoi problem is also used as a memory test by neuropsychologists for trying to evaluate **amnesia**

/* Write C programs that use recursive functions to solve Towers of Hanoi problem.*/

```
#include<conio.h>
#include<stdio.h>
#include<math.h>
void  hanoiRecursion( int num, char st, char dt, char it);
void main( )
{
        int no;unsigned long int n;
        clrscr( );
        printf("Enter the no. of disks to be transferred: ");
        scanf("%d",&no);
        if(no<1)
                printf("\nThere's nothing to move.");
        else
        {       n=pow(2,no)-1;
                printf("\nNumber of steps required for moving %d disks are: %lu",no,n);
                printf("\nEnter any key to display moves");
                getch( );
                printf("\nRecursive Function");
                printf("\n-------------------------------------");
                hanoiRecursion(no,'A','C','B');
        }
        getch( );
}
void  hanoiRecursion( int num,char st, char dt, char it)                /* Recursive Function*/
{
        if ( num == 1 )
        {
                printf("\nMove top disk from needle %c to needle %c.", st, dt);
                return;
        }
        hanoiRecursion( num - 1,st, it, dt );
        printf( "\nMove top disk from needle %c to needle %c.", st, dt );
```

```
            hanoiRecursion( num - 1,it, dt, st );
}
```

OUTPUT:

```
Enter the no. of disks to be transferred: 2
Number of steps required for moving 2 disks are: 3
Enter any key to display moves
Recursive Function
------------------------------
Move top disk from needle A to needle B.
Move top disk from needle A to needle C.
Move top disk from needle B to needle C.
```

OUTPUT:

```
Enter the no. of disks to be transferred: 3
Number of steps required for moving 3 disks are: 7
Enter any key to display moves
Recursive Function
-------------------------------
Move top disk from needle A to needle C.
Move top disk from needle A to needle B.
Move top disk from needle C to needle B.
Move top disk from needle A to needle C.
Move top disk from needle B to needle A.
Move top disk from needle B to needle C.
Move top disk from needle A to needle C.
```

UNIT-III

Program Structure: Storage classes, Automatic variables, External (global) variables, Static Variables, Multi file programs, more about library functions.

Arrays: Defining an array, processing an array, passing arrays to functions, Multi dimensional arrays.

Array Techniques: Array order reversal, Removal of duplicates from an ordered array, Finding the Kth smallest element.

Merging, Sorting and Searching: The two way merge, Sorting by selection, Sorting by exchange, Sorting by insertion, Sorting by partitioning, Recursive Quick sort, Binary Search.

Strings: Defining a string, NULL character, Initialization of strings, Reading and Writing a string, Processing the strings, Character arithmetic, Searching and Sorting of strings, Some more Library functions for strings

STORAGE CLASSES:

Data type refers to the type of information represented by a variable, e.g., integer number, floating-point number, character, etc. Storage class refers to the permanence of a variable, and its ***scope*** within the program, i.e., the portion of the program over which the variable is recognized.

If we do not specify the storage class of a variable in its declaration, the compiler will assume a storage class dependent on the context in which the variable is used. Thus C has got certain default storage classes.

The variables may also be categorized, depending on the place of their declaration, as INTERNAL (local) or EXTERNAL (global). Internal variables are within a particular function, while external variables are declared outside of any function.

From C compiler point of view, a variable name identifies some physical location within the computer's memory, where the strings of bits representing the variables value stored. There are some basically two kinds of locations in a computer where such a value may be kept: *memory* and *CPU register*. The variables storage class, which determines that in which of these two locations the value is stored.

Moreover, a variables storage class will gives information about:

- Where the variable would be stored. (i.e., location of the variable)
- What will be the initial value of the variable , if the initial value is not specifically assigned(i.e. the default initial value)
- What is the scope of the variable, i.e. in which function the value of the variable would be available.
- What is the life of the variable, i.e. how long would the variable exist.

TYPES OF STORAGE CLASSES IN C:

a) AUTOMATIC STORAGE CLASS.

b) REGISTER STORAGE CLASS.

c) STATIC STORAGE CLASS.

d) EXTERNAL STORAGE CLASS.

a) **AUTOMATIC STORAGE CLASS**:

- ***Automatic variables*** are always declared within a function and are local to the function in which they are declared; that is, their scope is confined to that function. Automatic variables defined in different functions will therefore be independent of one another, even though they may have the same name.
- Variables declared inside the function are precedes with the keyword '**auto**' and termed as automatic variables.

  Example: **auto int x;**
- By defining a variable as automatic storage class variable (i.e., Automatic variables), it is stored in the memory. Default value of automatic variable will be some garbage value.
- Scope of the variable is within the block where it is defined and the life time of the variable is until the control remains within the block. Because of this property, automatic variables are also referred to as local or internal variables.
- A variable declared inside a function without storage class specification is, by default, and automatic variable.
- *An automatic variable does not retain its value once control is transferred out of its defining function.* Therefore, any value assigned to an automatic variable within a function will be lost once the function is exited. If the program logic requires that an automatic variable be assigned a particular value each time the function is executed, that value will have to be reset whenever the function is reentered

Ex: /* Program to illustrate automatic variables */

```
#include<stdio.h>
#include<conio.h>
void function1(void);
void function2(void);
void main( )
{
        auto int m=1000;
        clrscr( );
        function2( );
        printf("%d \n",m);
        getch( );
}
void function1(void)
{
        auto int m=10;
        printf("%d\n",m);
}
void function2(void)
{
        auto int m=100;
        function1( );
        printf("%d\n",m);
}
```

OUTPUT:

```
10
100
1000
```

b) REGISTER STORAGE CLASS:

- A **register** storage class is the same as the auto class with only one difference. We can tell the compiler that a variable should be kept in one of the machine's register, instead of keeping in the memory. Since a register access is much faster than a memory access. Keeping the frequently accessed variables in the register will lead to faster execution of programs.
- When a variable is declared under ***register storage class***, it is stored in the CPU registers. To define a variable as register storage class, the keywords register. EG: **register int i;**
- Scope of the register variable is within the block where it is defined and the life time of the variable is until the control remains within the block.
- There is one restriction on the use of a register variable; a register variable address is not available to the user. This means that we can't use the address operator and the indirection operator with a register.

Eg: /* Program to declare a register variable */

```
#include<stdio.h>
#include<conio.h>
void main( )
{
     register int i=1;
     do
   {
              printf("%d\t",i);
                     i++;
   }while(i<=10);
    getch( );
}
OUTPUT:
1    2    3    4    5    6    7    8    9    10
```

c) STATIC STORAGE CLASS:

- As the name suggests, the value of static variables persists until the end of the program.
- A variable can be declared static using the keyword **static.**
- Static variable is stored in memory.
- A static variable may be either an internal type or an external type, depending on the place of declaration.
- **Internal static variables** are those which are declared inside a function. The scope of internal static variable extends up to the end of the function. Therefore internal static variables are similar to auto variables, except that they remain in existence (alive) throughout the remainder of the program.
- The default value of the static variable is zero.

Ex: /*program to illustrate the properties of static variables:*/

```
#include<stdio.h>
#include<conio.h>
void function(void);
void main( )
{
        static int i;
        static int j;
        clrscr( );
        printf("default value of int i=%d\t static j=%d\n",i,j);
        for(i=1;i<=3;i++)
                function( );
        getch( );
}
```

```
void function(void)

{

        static int x=1;

        printf("value of x after initialization =%d\n",x);

        x=x+3;

        printf("value of x after increment=%d\n", x);

}

OUTPUT:

default value of int i=0        static j=0

value of x after initialization =1

value of x after increment=4

value of x after initialization =4

value of x after increment=7

value of x after initialization =7

value of x after increment=10
```

- A static variable is initialized only once, when the program is compiled, it is never initialized again. During the first call to **function ( )**, **x** is incremented to 3.Because **x** is static, this value persists and therefore, the next call adds another 3 to x giving it a value of 7. The value of x becomes 10 when the third call is made.
- If the variable x in function ( ) is declared as int ; whenever the function ( ) is called the value is again initialized. The previous value doesn’t persist.
- **An external static variable** is declared outside of all functions and is available to all functions in that program. The difference between a static external variable and a simple external variable is that the static external variable is available only within the file where it is defined while the simple external variable can be accessed by other files.

d) EXTERNAL STORAGE CLASS:

- Variables that are both alive and active throughout the entire program are known as external variables. They are also known as global variables. Unlike local variables, global variables can be accessed by any function in the program.
- External variables are declared outside a function. When the variables are declared as an external by using keyword **extern,** it is stored in memory.
- External variable's default value is zero...

Below program is used to illustrate the properties of global variables. Note that variable **X** is used in all functions. But none except **function2** has a definition for **X.** Because **X** has been declared above all the functions, it is declared as global; any function can use it, and change its value. Then subsequent function can reference only those new values.

Ex: /* Program to illustrate the properties of global variables */

```
#include<stdio.h>
#include<conio.h>
int x;
int function1(void);
int function2(void);
int function3(void);
void main( )
{
        x=25;
        clrscr( );
        printf("value of x in main x=%d \n ",x);
        printf("value of x in function1 x=%d \n",function1( ));
        printf("value of x in function2 x= %d \n",function2( ));
        printf("value of x in function3 x=%d \n", function3( ));
        getch( );
}
```

```
int function1(void)
{
    x=x+10;
    return(x);
}
int function2(void)
{
    int x;
    x=10;
    return(x);
}
int function3( )
{
    x=x+10;
   return(x);
}
OUTPUT:
value of x in main x=25
 value of x in function1 x=35
value of x in function2 x= 10
value of x in function3 x=45
```

One other aspect of a global variable is that it is available only from the point of declaration to the end of the program. Consider a program segment as shown below:

```
main( )
{
        y= 5;
        . . . . .
        . . . . .
}
int y;          /*global declaration*/
```

```
func1( )
{
        y = y + 1;
}
```

We have a problem here. As far as main is considered, **y** is not defined. So, the compiler will issue an error message. Unlike local variables, global variables are initialized to zero by default. The statement y = y + 1; in **fun1** will, therefore, assign 1 to y.

External Declaration:

In the program segment above, the **main** cannot access the variable y as it has been declared after the **main** function. This problem can be solved by declaring the variable with the storage class **extern.**

For example:

```
main ( )
{
        extern int y;
        . . . . . . . . .             /*global declaration*/
        . . . . . . . . .
}
func1( )
{
        extern int y;                 /*external declaration*/
        . . . . . . . . .
        . . . . . . . . .
}
int y;                                /* declaration*/
```

Although the variable ***y*** has been defined after both the functions, the external declaration of ***y*** inside the functions informs the compiler that ***y*** is an integer type defined somewhere else in the program. Note that **extern** declaration does not allocate storage space for variables. In case of arrays, the definition should include their size as well.

SCOPE RULES:

SCOPE: The region of a program in which a variable is available for use

VISIBILITY: The program’s ability to access a variable from the memory

LIFETIME: The lifetime of a variable is the duration of time in which a variable exists in the memory during execution.

Rules of use:

1. The scope of a global variable is the entire program file.
2. The scope of a local variable begins at point of declaration and ends at the end of the block or function in which it is declared.
3. The scope of a formal function argument is its own function.
4. The life time (or longevity) of an **auto** variable declared in **main** is the entire program execution time, although its scope is only **main** function.
5. The life of an **auto** variable declared in a function ends when the function is exited.
6. A **static** local variable, although its scope is limited to its function, its lifetime extends till the end of program execution.
7. All variables have visibility in their scope, provided they are not declared again.
8. If a variable is re-declared within its scope again, it loses its visibility in the scope of the re-declared variable.

TYPE QUALIFIERS:

C defines type qualifiers that control how variables may be accessed or modified. C provides three types of type qualifiers:

i) const ii) volatile iii) restrict

const & volatile qualifiers can be applied to any variables, but *restrict* qualifiers may only applied to pointer variables.

i) Constant Variable:

A variable value can be made unchanged during program execution by declaring the variable as a constant. The keyword **const** is placed before the declaration of a variable.

For a constant pointer, place the keyword **const** between * and identifier**.**

Syntax: **const data_type var_name;**

Eg 1: const int a; here, **a** is a constant and it's value cannot be changed.

Eg 2: int * const x; Here, the pointer to **x** is constant. The value that **x** points can be changed, but the value of x cannot be changed.

Eg 3: const int *x; (or) int const *x;

In the above example, both the statements give the same meaning. The value that **x** points to is a constant integer and cannot be changed. However, the value of **x** can be changed.

ii) Volatile Variable:

Variables that can be changed at any time by external programs or the same program are called as volatile variables. The keyword **volatile** is placed before the declaration of the variable.

To make a variable value changeable by the current program and unchangeable by other programs, declare the variable as **volatile** and **constant.**

Syntax: volatile int x; volatile const int y; volatile int * z; (or) int * volatile z;

The variable **x** value can be changed by any program at any time and the variable **y** can be changed by the current program but not by the external programs. The variable **z** is a pointer to a volatile integer.

iii) Restrict Variable:

The restrict type qualifier may only be applied to a pointer. A pointer declaration that uses this type qualifier establishes a special association between the pointer and the object it accesses, making that pointer and expressions based on that pointer is the only way to directly or indirectly access the value of that object.

Generally, a pointer is a variable which is used to store the address of the other variable. More than one pointer can access the same chunk of memory and modify it during the execution of the program.

The **restrict** type qualifier is an indication to the compiler that, if the memory addressed by the **restrict-**qualified pointer is modified, then no other pointer will access the same memory.

MULTIFILE PROGRAMS:

- ***A file*** is a collection of information stored as a separate entity within the computer or on an auxiliary storage device.
- **A** file can be a collection of data, a source program, a portion of a source program, an object program, etc.
- Usually ***C*** programs are contained entirely within a single file. Many programs, however, are composed of multiple files. This is especially true of programs that make use of lengthy functions, where each function may occupy a separate file. Or, if there are many small related functions within a program, it may be desirable to place a few functions within each of several files.
- The individual files will be compiled separately, and then linked together to form one executable object program
- Multi file programs allow greater flexibility in defining the scope of both functions and variables.
- The rules associated with the use of storage classes become more complicated, however, because they apply to functions as well as variables, and more options are available for both external and static variables.

Functions

- Within a multi file program, a function definition may be either ***external*** or ***static.*** An external function will be recognized throughout the entire program, whereas a static function will be recognized only within the file in which it is defined.
- In each case, the storage class is established by placing the appropriate storage-class designation (i.e., either **ex**t**e**rn or **static)** at the beginning of the function definition. The function is assumed to be ***external*** if a storage class designation does not appear. In general terms, the first line of a function definition can be written as
  storage-class data-type funname( );
- When **a** function is defined in one file and accessed in another, the latter file must include a function ***declaration.***

- To execute a multifile program, each individual file must be compiled and the resulting object files linked together. To do so, we usually combine the source files within a ***project.*** We then ***build*** the project (i.e., compile all of the source files and link the resulting object files together into a single executable program). If some of the source files are later changed, we ***make*** another executable program (i.e., compile the new source files and link the resulting object files, with the unchanged object files, into a new executable program).

Consider an example in which a C program is written in two different files and save them as file1.c and file2.c.

FIRST FILE (file1.c)

```
#include<stdio.h>

#include<conio.h>

extern void output(void); / * function prototype */

void main( )

{

        clrscr( );

        output ( ) ;

        getch( );

}
```

SECOND FILE (file2.c)

```
extern void output(void)

{

        printf (“Hello, knreddy “) ;

}
```

After creating two files we select **open project** from project menu and specify **projectname.ide** as project name. This will result in the Project window being opened. Then add two files using **add item** from project menu.

The program can now be executed by selecting Run from the Debug menu.

If a file contains a static function, it may be necessary to include the storage class **static** within the function declaration or the function prototype.

Variables

- Within a multifile program, external (global) variables can be defined in one file and accessed in another.
- An external variable ***definition*** can appear in only one file. Its location within the file must be external to any function definition. Usually, it will appear at the beginning of the file, ahead of the first function definition.
- In order to access an external variable in another file, the variable must first be ***declared*** within that file.
- This declaration may appear anywhere within the file. Usually, however, it will be placed at the beginning **of** the file, ahead of the first function definition.
- The declaration ***must*** begin with the storage-class specifier **extern.**

<u>FIRST FILE (file1.c)</u>

```
#include<stdio.h>
#include<conio.h>
int x=5,y=10;        /* external variables*/
extern void output(void); / * function prototype */
void main( )
{
        clrscr( );
        output ( ) ;
        getch( );
}
```

<u>SECOND FILE (file2.c)</u>

```
extern int x,y;
extern void output(void)
{
        printf (“x=%d\ny=%d”,x,y) ;
}
```

Here the output will be x=5   y=10 as x and y are external variables.

- Within a file, external variables can be defined as static. To do **so,** the storage-class specifier **static** is placed at the beginning of the definition. The scope of a static external variable will be the remainder of the file in which it is defined. It will not be recognized elsewhere in the program (Le, in other files). Thus, the use of static external variables within **a** file permits a group of variables to be “hidden” from the remainder of a program.

  If we write static in the second file as :

  <u>SECOND FILE (file2.c)</u>

```
static  int x,y;
extern void output(void)
{
        printf (“x=%d\ny=%d”,x,y) ;
}
```

  output is

```
x=0
y=0
```

  As x and y are declared as static these are recognized only within the second file.

ARRAYS:

C programming language allows working with collection of data elements of same data type called as an array. The elements of an array are stored sequentially in the memory. In C strings are stored as an array of characters.

Introduction:

In many applications, we need to handle a large volume of data in terms of reading, processing and printing. To process such large amounts of data, we need a powerful data type that would facilitate efficient sorting, accessing and manipulation of data items.

C supports a derived data type known as **array** that can be used for such applications.

"An array is a fixed-size sequenced collection of elements of the same data type. It is simply a grouping of like-type data".

An array is a sequenced collection of related data that share a common name.

An array is referred by specifying the array name followed by one or more subscripts, with each subscript is enclosed in square brackets. The number of subscripts determines the dimensionality of the array. Depending on the number of subscripts used, arrays can be classified into different types as:

1. One dimensional arrays
2. Two dimensional arrays
3. Multi dimensional arrays

Each array element (i.e., each individual data item) is referred to by specifying the array name followed by one or more ***subscripts,*** with each subscript enclosed in square brackets. Each subscript must be expressed as a nonnegative integer. In an n-element array, the array elements are **x [0] , x [1] , x[2], . . . ,x [n - 1]**

Properties of an Array:

- An array is a collection of similar elements.
- The first element in the array is numbered 0, so the last element is 1 less than the size of the array
- An array is also known as a subscripted variable.
- Before using an array its type and dimension must be declared.
- However big an array its elements are always stored in contiguous memory locations
- The type of an array is the data type of its elements.
- The location of an array is location of its first element.
- The length of an array is the number of data elements in the array.
- The size of an array is the length of the array times the size of an element.

Array whose element are specified by one subscript are called single subscripted or single dimensional array. Analogous array whose elements are specified by two and three subscripts are called two-dimensional or double subscripted and three-dimensional or triple-subscripted arrays respectively.

ONE-DIMENSIONAL ARRAYS

Like any other variable, arrays must be declared before they are used so that the compiler can allocate space for them in the memory.

The general form of ARRAY DECLARATION is

Syntax: **data_type variable-name [size] ; /* uninitialized*/**

data_type variable-name [size]={initialized list}; /*initialized*/

The type specifies the type of element that will be contained in the array, such as int, float, or char and the size indicates the maximum number of elements that can be stored in the array. For example, float height [50]; declares the height to be an array containing 50 real elements. Any subscripts 0 to 49 are valid. Similarly, int group [10]; declares the group as an array to contain a maximum of 10 integer constants.

The general uninitialized array declaration just allocates storage space for array consisting of size of memory cells. Each memory cell store one data item whose data type is specified.

ARRAY SUBSCRIPT: arrayname[subscript]

The subscript may be any expression of type int. Each time a subscripted variable is encountered in a program, the subscript is evaluated and its value determines which element of array is referenced.

Remember the following points:

It is programmer's responsibility to verify that the subscript is within the declared range. Any reference to the arrays outside the declared limits would not necessarily cause an error. Rather, it might result in unpredictable program results.

The size should be either a numeric constant or a symbolic constant.

The 'C' language treats character strings simply as arrays of characters. The size in a character string represents the maximum number of characters that the string can hold. For instance, char name[10] ; declares the name as a character array (string) variable that can hold a maximum of 10 characters. Suppose we read the following string constant into the string variable name. "WELL DONE"

Each character of the string is treated as the element of the array name and is stored in the memory as follows:

'W'	'E'	'L'	'L'	' '	'D'	'O'	'N'	'E'	'\0'

When the compiler sees a character string, it terminates it with an additional null character. Thus, the element name [10] holds the null character '\0'. When declaring character arrays, we must allow one extra element space for the null terminator.

Initialization of One-dimensional arrays:

After an array is declared, its elements must be initialized. Otherwise, they will contain "garbage". An array can be initialized at either of the following stages:

At compile time

At run time

i) Compile Time Initialization

We can initialize the elements of arrays in the same way as the ordinary variables when they are declared. The general form of initialization of array is:

```
datatype  array-name [size]  =  { list of values };
```

The values in the list are separated by commas. For example, the statement

```
int  number [3]  =  { 1,2,3 } ;
```

This will declare the variable number as an array of size 3 and will assign 1to first element , 2 to second element and 3 to the third element.

1	2	3
number [0]	number[1]	number[2]

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

```
float  total[5]  =  { 0.0 , 15.75 , -10 };
```

This will initialize the first three elements 0.0, 15.75 and -10.0 and the remaining two elements to zero.

0.0	15.75	-10	0	0
total[0]	total[1]	total[2]	total[3]	total[4]

The size may be omitted. In such cases, the compiler allocates enough space for all initialized elements.

For example, the statement

```
int  counter[ ]  =  {1,1,1,1 } ;
```

This will declare the counter array to contain four elements with initial values 1. This approach works fine as long as we initialize every element in the array.

Character arrays may be initialized in a similar manner. Thus, the statement

```
char  name[ ]  =  {  ‘k’ , ’n’ , ’r’ , ’e’ , ‘d’,’d’,’y’,’\0’ } ;
```

declares the name to be an array of eight characters, initialized with the string “knreddy” ending with the null character. Alternatively, we can assign the string literal directly as under:

```
char name [ ]  =  “knreddy”;
```

Compile time initialization may be partial. That is, the number of initializers may be less than the declared size. In such cases, the remaining elements are initialized to zero, if the array type is numeric and NULL if the type is char. For example,

```
int  number [5]  =  { 10, 20 };
```

will initialize the first two elements to 10 and 20 respectively, and the remaining elements to 0.

10	20	0	0	0
number [0]	number [1]	number [2]	number [3]	number [4]

Similarly the declaration

```
char  city [5]  =  { ‘K’ };
```

will initialize the first element to ‘K’ and the remaining four to NULL.

‘B’	‘\0’	‘\0’	‘\0’	‘\0’
city [0]	city [1]	city [2]	city [3]	city [4]

It is a good idea, however, to declare the size explicitly, as it allows the compiler to do some error checking.

Remember, however, if we have more initializers than the declared size, the compiler will produce an error. That is, the statement int number [3] = { 10, 20, 30, 40 } ; will not work. It is illegal in C.

ii) Run Time Initialization

An array can be explicitly initialized at run time. This approach is usually applied for initializing large arrays.

For example, consider the following segment of a C program.

```
for ( i = 0 ; i < 100 ; i = i+1 )
{
    if   i <  50
          sum[i] = 0.0 ;                    /* assignment statement */
    else
          sum[i] = 1.0 ;
}
```

The first 50 elements of the array sum are initialized to zero while the remaining 50 elements are initialized to 1.0 at run time.

We can also use a read function such as scanf to initialize an array. For example, the statements

```
int  x [3] ;
scanf (" %d%d%d ", &x[0], &x[1], &x[2] ) ;
```

will initialize array elements with the values entered through the keyboard.

PROCESSING AN ARRAY:

Single operations which involve entire arrays are not permitted in C. Thus, if a and b are similar arrays (i.e., same data type, same dimensionality and same size), assignment operations, comparison operations, etc. must be carried out on an element-by-element basis. This is usually accomplished within a loop, where each pass through the loop is used to process one array element. The number of passes through the loop will therefore
equal the number of array elements to be processed.

Very often, we wish to process the elements of an array in sequence, starting with elements zero. An example would be scanning data into the array or printing its contents. In C, we can accomplish this processing easily using an indexed for loop, a counting loop whose loop control variable runs from zero to one less than the array size. Using the loop counter as an array index(subscript) gives access to each array element in turn.

EG:Write a C program to read any ten numbers and store them in an array. Display all the elements in the array and find the total of all elements in the array.

```
#include<stdio.h>
#include<conio.h>
void main ( )
{
        int  i;
        float  x[10], value, total;
        clrscr( );
        /*……..READING VALUES INTO AN ARRAY……..*/
        printf("Enter 10 REAL NUMBERS\n");
        for( i =0 ; i<10 ; i++)
        {
                printf("enter number %d  :",i+1);
                scanf("%f", &value) ;
                x[i] = value ;
        }
        /*……….. COMPUTATION OF TOTAL ………*/
        total = 0.0 ;
                for( i=0 ; i< 10 ; i++)
                total = total + x[i] ;
/*….PRINTING OF x[i] VALUES AND TOTAL …...*/
             printf(" \n") ;
            for( i = 0 ; i < 10 ; i++)
                        printf("\nx[%2d] = %5.2f", i , x[i] ) ;
             printf("\ntotal = %.2f\n ",total) ;
            getch( );
}
```

OUTPUT:

```
Enter 10 REAL NUMBERS
enter number 1  :2
enter number 2  :4
enter number 3  :6
enter number 4  :8
enter number 5  :10
enter number 6  :12
enter number 7  :14
enter number 8  :16
enter number 9  :18
enter number 10  :20

x[ 0] =  2.00
x[ 1] =  4.00
x[ 2] =  6.00
x[ 3] =  8.00
x[ 4] = 10.00
x[ 5] = 12.00
x[ 6] = 14.00
x[ 7] = 16.00
x[ 8] = 18.00
x[ 9] = 20.00
total = 110.00
```

EXAMPLE PROGRAMS –ARRAYS

Ex: 1/* Program to read and print the salaries of 10 employees */

```
#include<stdio.h>
#include<conio.h>
void main( )
{
        int sal[10],i;
        clrscr( );
        printf("enter salaries of ten employees\n");
        for(i=0;i<10;i++)
        {
                printf("\nEnter the salary of employee %d : ",i+1);
                scanf("%d",&sal[i]);
        }
        for(i=0;i<10;i++)
                printf("\nsal[%d]=%d",i+1,sal[i]);
        getch( );
}
```

OUTPUT:

```
enter salaries of ten employees
Enter the salary of employee 1 : 300
Enter the salary of employee 2 : 200
Enter the salary of employee 3 : 4530
Enter the salary of employee 4 : 5000
Enter the salary of employee 5 : 7452
Enter the salary of employee 6 : 5423
Enter the salary of employee 7 :1200
Enter the salary of employee 8 : 4500
Enter the salary of employee 9 : 8000
Enter the salary of employee 10 : 4500

sal[1]=300
sal[2]=200
sal[3]=4530
sal[4]=5000
sal[5]=7452
sal[6]=5423
sal[7]=1200
sal[8]=4500
sal[9]=8000
sal[10]=4500
```

```
Ex 2:/* Program to find minimum and maximum values of array */
#include<stdio.h>
#include<conio.h>
void main( )
{
        int a[10],min,max,i;
        clrscr( );
        printf("enter ten elements of array \n");
        for(i=0;i<10;i++)
                scanf("%d",&a[i]);
        min=a[0];
        max=a[0];
        for(i=0;i<10;i++)
        {
                if(min>a[i])
                        min=a[i];
                if(max<a[i])
                        max=a[i];
        }
        printf("\nminmum value :%d",min);
        printf("\nmaximum value:%d",max);
        getch( );
}
```

OUTPUT:

```
enter ten elements of array
5
12
63
45
12
8
44
78
24
12
minmum value :5
maximum value:78
```

Ex 3: /* Program to sort the elements of an array in ascending order */

```
#include<stdio.h>
#include<conio.h>
void main( )
{
        int a[10],i,j,n=10,temp;
        clrscr( );
        printf("enter ten elements of the array\n");
        for(i=0;i<n;i++)
        scanf("%d",&a[i]);
        printf("elements of the array before sorting\n");
        for(i=0;i<n;i++)
        printf("a[%d]=%d\t",i,a[i]);
        for(i=0;i<n-1;i++)
        {
                for(j=0;j<n-1-i;j++)
                {
                        if(a[j]>a[j+1])
                        {
                                temp=a[j];
                                a[j]=a[j+1];
                                a[j+1]=temp;
                        }
                }
        }
        printf("\nelements of the array after sorting\n");
        for(i=0;i<n;i++)
        printf("a[%d]=%d\t",i,a[i]);
        getch( );
}
```

OUTPUT:

```
enter ten elements of the array
2       3       4       1       5       9       8       7       6       10
elements of the array before sorting
a[0]=2  a[1]=3  a[2]=4  a[3]=1  a[4]=5  a[5]=9  a[6]=8  a[7]=7  a[8]=6  a[9]=10
elements of the array after sorting
a[0]=1  a[1]=2  a[2]=3  a[3]=4  a[4]=5  a[5]=6  a[6]=7  a[7]=8  a[8]=9  a[9]=10
```

```
Ex 4: /* program to swap of two arrays */
#include<stdio.h>
#include<conio.h>
void main( )
{
        int a[5],b[5],i;
        clrscr( );
        printf("enter the elements of first array\n");
        for(i=0;i<5;i++)
                scanf("%d",&a[i]);
        printf("enter the values of second array\n");
        for(i=0;i<5;i++)
                scanf("%d",&b[i]);
        printf("elements of two arrays before swapping \n");
        printf("\nFirst array elements \n");
        for(i=0;i<5;i++)
                printf("a[%d]=%d\t",i , a[i]);
        printf("\nSecond array elements \n");
        for(i=0;i<5;i++)
                printf("b[%d]=%d\t",i, b[i]);
        for(i=0;i<5;i++)
        {
                a[i]=a[i]+b[i];
                b[i]=a[i]-b[i];
                a[i]=a[i]-b[i];
        }
        printf("\nelements of two arrays after swapping \n");
        printf("\nFirst array elements \n");
        for(i=0;i<5;i++)
                printf("a[%d]=%d\t",i , a[i]);
        printf("\nSecond array elements \n");
        for(i=0;i<5;i++)
                printf("b[%d]=%d\t",i, b[i]);
        getch( );
}
```

OUTPUT:

```
enter the elements of first array
1
2
3
4
5
enter the values of second array
6
7
8
9
10
elements of two arrays before swapping

First array elements
a[0]=1  a[1]=2  a[2]=3  a[3]=4  a[4]=5
Second array elements
b[0]=6  b[1]=7  b[2]=8  b[3]=9  b[4]=10
elements of two arrays after swapping

First array elements
a[0]=6  a[1]=7  a[2]=8  a[3]=9  a[4]=10
Second array elements
b[0]=1  b[1]=2  b[2]=3  b[3]=4  b[4]=5
```

MULTIDIMENSIONAL ARRAY

A multidimensional array has more than one subscript. A two-dimensional array has two subscripts; a three-dimensional array has three subscripts, and so on. There is no limit to the number of dimensions a C array can have.

Declaration:

The two-dimensional array can be declared by specifying first the base type of the array, then the name of the array variable, and then the number of rows and column elements the array will have should be specified between a pair square brackets ([ ] [ ]). Note that this value cannot be a variable and has to be an integer constant.

General Format:

```
data_type  arrayname [  ] [   ];
```

Array initialization:

The elements of a two-dimensional array will be assigned by rows; i.e., the elements of the first row will be assigned, then the elements of the second row, and so on.

Elements of an array can be assigned initial values by following the array definition with a list of initializes enclosed in braces and separated by comma.

int array[4][3] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12 };

results in the following assignments:

array[0][0] is equal to 1	array[0][1] is equal to 2	array[0][2] is equal to 3
array[1][0] is equal to 4	array[1][1] is equal to 5	array[1][2] is equal to 6
array[2][0] is equal to 7	array[2][1] is equal to 8	array[2][2] is equal to 9
array[3][0] is equal to 10	array[3][1] is equal to 11	array[3][2] is equal to 12

When you initialize multidimensional arrays, you can make your source code clearer by using extra braces to group the initialization values and also by spreading them over several lines. The following initialization is equivalent to the one just given:

```
int array[4][3] = {     { 1, 2, 3 } ,
                        { 4, 5, 6 } ,
                        { 7, 8, 9 } ,
                        { 10, 11, 12 }
                  };
```

Remember, initialization values must be separated by a comma--even when there is a brace between them. Also, be sure to use braces in pairs--a closing brace for every opening brace--or the compiler becomes confused

Ex 1: /* program to print the numbers in matrix format using two dimensional arrays */

```
#include<stdio.h>
#include<conio.h>
main()
{
        int matrix[4][3],i,j;

        printf("enter the numbers in row wise \n");
        for(i=0;i<4;i++)
        {
           for(j=0;j<3;j++)
              scanf("%d",&matrix[i][j]);
        }
        printf("\nThe entered numbers in matrix format :");
        for(i=0;i<4;i++)
        {
           printf("\n\n");
           for(j=0;j<3;j++)
              printf("%d\t",matrix[i][j]);
        }
        getch( ) ;
}
```

OUTPUT:

```
enter the numbers in row wise
1       2       3       4       5       6       7       8       9       10      11      12
The entered numbers in matrix format
1     2     3
4     5     6
7     8     9
10     11     12
```

Ex 2: /*Program to compute and print multiplication table using arrays*/

```
#include<stdio.h>
#include<conio.h>
#define rows 10
#define columns 10
void main( )
{
        int row,col,product[rows][columns];
        int i,j;
        clrscr( );
        printf("multiplication table \n\n");
        printf("  * ");
        for(j=1;j<=columns;j++)
                printf("%4d",j);
        printf("\n");
        printf("............................................\n");
        for(i=0;i<rows;i++)
        {
                row=i+1;
                printf("%4d",row);
                for(j=1;j<=columns;j++)
                {
                        col=j;
                        product[i][j]=row*col;
                        printf("%4d", product[i][j]);
                }
                printf("\n");
        }
        getch( );
}
```

OUTPUT:

```
multiplication table


 *    1     2     3     4     5     6     7     8     9     10
.......................................................................................................................
  1   1     2     3     4     5     6     7     8     9     10
  2   2     4     6     8     10    12    14    16    18    20
  3   3     6     9     12    15    18    21    24    27    30
  4   4      8    12    16    20    24    28    32    36    40
  5   5     10    15    20    25    30    35    40    45    50
  6   6     12    18    24    30    36    42    48    54    60
  7   7     14    21    28    35    42    49    56    63    70
  8   8     16    24    32    40    48    56    64    72    80
  9   9     18    27    36    45    54    63    72    81    90
 10   10    20    30    40    50    60    70    80    90    100
```

Ex 3: /*Write a program to print identity matrix */

```
#include<stdio.h>
#include<conio.h>
void  main( )
{
        int i,j,row,col,a[10][10];
        clrscr( );
        printf("enter number of rows and columns");
        scanf("%d%d",&row,&col);
        if(row!=col)
                printf("Identity matrix is not possible");
        else
        {
                for(i=0;i<row;i++)
                {
                        for(j=0;j<col;j++)
                        {
                                if(i==j)
                                        a[i][j]=1;
                                else
                                        a[i][j]=0;
                        }
                }
                for(i=0;i<row;i++)
                {
                        for(j=0;j<col;j++)
                        {
                                printf("%3d",a[i][j]);
                        }
                        printf("\n");
                }
        }
        getch( );
}
```

OUTPUT:

```
enter number of rows and columns 3 3
  1  0  0
  0  1  0
  0  0  1
```

Ex4:/*Write a program to print pascal's triangle using arrays*/

```
#include<stdio.h>
#include<conio.h>
void main( )
{
        int i,j,position=30,line,a[20][20];
        clrscr( );
        printf("Enter number of lines to print pascal triangle\n");
        scanf("%d",&line);
        for(i=0;i<line;i++)
        {
                gotoxy(position,i+2);
                for(j=0;j<=i;j++)
                {
                        if(j==0||i==j)
                                a[i][j]=1;
                        else
                                a[i][j]=a[i-1][j-1]+a[i-1][j];
                        printf("%6d",a[i][j]);
                }
                position=position-3;
                printf("\n\n");
        }
        getch( );
}
```

OUTPUT:

```
Enter number of lines to print pascal triangle
5                    1
                  1     1
               1     2     1
            1     3     3     1
         1     4     6     4     1
```

Write a program to convert a given decimal to its equivalent binary number

```
#include<stdio.h>
#include<conio.h>
void main( )
{
        int n,i,j,x[10];
        clrscr( );
        printf("\nEnter a Decimal Number:");
        scanf("%d",&n);
        i=0;
        while(n>0)
        {
                x[i]=n%2;
                i++;
                n=n/2;
        }
        printf("\nBINARY EQUIVALENT IS:");
        for(j=i-1;j>=0;j--)
                printf("%5d",x[j]);
        getch( );
}
```

OUTPUT:

```
Enter a Decimal Number:8
BINARY EQUIVALENT IS:    1    0    0    0
```

Eg: An election is conducted by 5 candidates. They are numbered 1-5 and voting is done by making the candidate number on the ballet paper. Write a program to read the ballets and count the votes for each candidate using an array variable count. In case a number read it outside the range 1-5 and the ballet is consider invalid and count them.

```
#include<stdio.h>
#include<conio.h>
void main( )
{
    int i, voter,s=0,ch;

    int vote[5]={0,0,0,0,0};
    clrscr( );
    printf("Enter number of voters");
    scanf("%d",&voter);
    for(i=1;i<=voter;i++)
    {
      printf("vote for any one of the candidate\n");
      printf("enter your choice from 1 to 5\n");
      scanf("%d",&ch);
      switch(ch)
      {
        case 1: vote[0]++;
                break;
        case 2: vote[1]++;
                break;
        case 3: vote[2]++;
                break;
        case 4: vote[3]++;
                break;
        case 5: vote[4]++;
                break;
        default: s++;
      }
    }
      printf("number of votes for 1st person=%d\n",vote[0]);
      printf("number of votes for 2nd person=%d\n",vote[1]);
      printf("number of votes for 3rd person=%d\n",vote[2]);
      printf("number of votes for 4th person=%d\n",vote[3]);
      printf("number of votes for 5th person=%d\n",vote[4]);
      printf("number of invalid votes =%d",s);
      getch( );
}
```

OUTPUT:

```
Enter number of voters5
vote for any one of the candidate
enter your choice from 1 to 5
1
vote for any one of the candidate
enter your choice from 1 to 5
2
vote for any one of the candidate
enter your choice from 1 to 5
1
vote for any one of the candidate
enter your choice from 1 to 5
2
vote for any one of the candidate
enter your choice from 1 to 5
8
number of votes for 1st person=2
number of votes for 2nd person=2
number of votes for 3rd person=0
number of votes for 4th person=0
number of votes for 5th person=0
number of invalid votes =1
```

PASSING ARRAYS TO FUNCTIONS

USING ARRAY ELEMENTS AS FUNCTION ARGUMENTS

We can use array elements as function arguments. For example in the call printf("%d",x[i]); array element x[i] is used as input argument in the printf function. When i is 3 them value x[3] is passed to printf and displayed.

Similarly in the call scanf("%d",&x[i]); array element x[i] is used as output argument in the scanf function. When i is 3 the address of x[3] is passed to scanf stores the scanned values in element x[3].

We can pass array elements as arguments to user defined functions. This can be done either by passing their data values or by passing their addresses.

Passing data values:

We can pass data values that are individual array elements, just like we pass any data value. As long as the array element type matches the function parameter type, it can be passed. The called function cannot tell whether the value it receives comes from an array, variable, or an expression.

passing variable	passing array element

main( )
{
int a;
fun(a); /*function call with a as argument*/
}
void fun(int x) /* function definition*/
{
}
 	
	main( )
	{
	int array[10];
	fun(array[3]); /*function call with array[3] element as argument*/
	}
	void fun(int x) /* function definition*/
	{
	}

In the above example the formal argument in function definition gets the value of a in first program .Similarly in the second program x gets the value of the element array [3].

Passing addresses:

We can pass the address of an individual element in the array just like we can pass the address of any variable. To pass an array element's address, we prefix the address operator to the element's indexed reference.

Passing an address of an array element requires two changes in the called function. First, it must declare that it is receiving an address. Second, it must use the indirection operator (*) to refer to the elements value.

passing address of variable	passing address of array element

main( )
{
   int a;
   fun(&a); /*function call with a as argument*/
}
void fun(int *x) /* function definition*/
{
 	
}
	main( )
	{
	   int array[10];
	fun(&array[3]); /*function call with array[3] element as argument*/
	}
	void fun(int *x) /* function definition*/
	{
	
	}

Array elements can be passed to a function by calling the function:

By value i.e. by passing values of array elements to the function.

By reference i.e. passing addresses of array elements to the function

/*Program to the accept a static array and print it by call by value*/

	OUTPUT:

#include<stdio.h>
#include<conio.h>
void write(int);
void main( )
{
    int i;
    int a [5]={33, 44, 55, 66, 77};
    clrscr( );
    for (i=0;i<5;i++)
    write (a[i]);
    getch( );
}
void write (int n)
{
   printf ("%d\n", n);
}
	33
	44
	55
	66
	77

/*Program to accept a static array and print it by call by reference */

```
#include<stdio.h>
#include<conio.h>
void write(int );
void main( )
{
    int i;
    int a[5]={33, 44, 55, 66, 77};
    clrscr( );
    for (i=0; i<5; i++)
    write (&a[i]);
    getch( );
}
void write(int *n)
{
   printf ("%d\n", *n);
}
```
	OUTPUT:
	33
	44
	55
	66
	77

ARRAY ARGUMENTS

Besides passing individual array elements to functions, we can write functions that have arrays as arguments. Such functions can manipulate some or all of the elements corresponding to an actual array argument.

When an array with no subscript appears in the argument list of a function call, what is actually stored in the function’s corresponding formal parameter is the address of the initial array element. In the function body, we can use subscripts with the formal parameters to access the array’s element. However, the function manipulates the original array, not its own personal copy, so an assignment to one of the array elements by a statement in the function changes the contents of the original array.

Fixed length arrays:

To pass the whole array we, simply use the array name as the actual parameter. In the called function, we declare that the corresponding formal parameter is an array. We do not need to specify the number of elements in a fixed-length array.

The function declaration for an array can declare the array in two ways. First, it can use an array declaration with an empty set of brackets in the parameter list. Second, it can specify that the array parameter is a pointer.

```
void fun(int array[ ],…….);     ⎫
void fun(int *array,……..);      ⎬     function declaration
                                ⎭
```

Variable- length arrays:

When the called function receives a variable-length array, we must declare and define it as variable length. In a function declaration, we declare an array as variable using an asterisk (*) for the array size or by using a variable for the array size. In the function definition, we must use the variable name, and following standard syntax rules, it must be defined before the array.

```
float fun(int size, float[ * ]);
```

RULES TO PASS THE WHOLE ARRAY TO A FUNCTION:

The function must be called passing only the name of the array.

In the function definition, the formal parameter must be an array type, either fixed length or variable length.

The size of fixed length array does not to be specified.

The size of variable length array in the function prototype must be an asterisk (*) or variable.

The size of variable length array in the function definition must be a variable that is in the function's scope, as a previously specified variable parameter.

TO PASS TWO-DIMENSIONAL ARRAY TO A FUNCTION:

The function must be called passing only the name of the array.

In the function definition, the formal parameter is a two-dimensional array, with the size of the second dimension required for a fixed-length array.

In the function definition for a variable length array, the size of all dimensions must be specified.

Passing an array as a constant:

When a function receives an array and doesn't change it, the array should be received as a constant array. This prevents the function from accidentally changing the array. To declare an array as a constant, we prefix its type with the type qualifier const, as shown in the following function declaration. Once an array is declared constant, the compiler will flag any statement that tries to change it as an error.

```
double average(const int array[ ], int size);
```

PASSING AN ENTIRE ARRAY TO A FUNCTION:

Let us now see how to pass the entire array to a function rather than individual elements.

Consider the following example:

```
#include<stdio.h>
#include<conio.h>
void display(int *, int);
void main( )
{
     int a[]={32, 43, 54, 65, 76};
     clrscr( )
     display(&a[0],5);   /*display(a,5);*/
     getch( );
}
void display(int *i, int x)
{
     int j;
     for (j=0; j<5;j++)
     {
               printf ("address=%u\t", i);
              printf ("element=%d\n",*i);
               i++;
     }
}
```

OUTPUT:

```
address=2293536    element=32
address=2293538    element=43
address=2293540    element=54
address=2293542    element=65
address=2293544    element=76
```

```
/*Write a program to print addition of given two matrices.*/
#include<stdio.h>
#include<conio.h>
void main( )
{
        int m,n,p,q,i,j,x[10][10],y[10][10],z[10][10];
        clrscr( );
        printf("\nEnter row size of Matrix 1:");
        scanf("%d",&m);
        printf("\nEnter column size of Matrix 1:");
        scanf("%d",&n);
        printf("\nEnter row size of Matrix 2:");
        scanf("%d",&p);
        printf("\nEnter column size of Matrix 2:");
        scanf("%d",&q);
        if(m==p && n==q)
        {
                printf("\nEnter Matrix 1 Elements:");
                for(i=0;i<m;i++)
                {
                        for(j=0;j<n;j++)
                        scanf("%d",&x[i][j]);
                }
                printf("\nEnter Matrix 2 Elements:");
                for(i=0;i<p;i++)
                {
                        for(j=0;j<q;j++)
                        scanf("%d",&y[i][j]);
                }
                for(i=0;i<m;i++)
                {
                        for(j=0;j<n;j++)
                        z[i][j]=x[i][j]+y[i][j];
                }
                printf("\nAddition Matrix Elements Are:");
                for(i=0;i<m;i++)
                {
                        printf("\n");
                        for(j=0;j<n;j++)
                        printf("%5d",z[i][j]);
                }
        }
        else
                printf("\nMATRIX ADDITON NOT POSSIBLE");
        getch( );
}
```

OUTPUT:

```
Enter row size of Matrix 1:2

Enter column size of Matrix 1:2

Enter row size of Matrix 2:2

Enter column size of Matrix 2:2

Enter Matrix 1 Elements:1 1 1 1

Enter Matrix 2 Elements:2 2 2 2

Addition Matrix Elements Are:
   3    3
   3    3
```

```
/*Write a program to print multiplication of given two matrices.*/
#include<stdio.h>
#include<conio.h>
void main( )
{
	int m,n,p,q,i,j,k,x[10][10],y[10][10],z[10][10];
	clrscr( );
	printf("\nEnter row size of Matrix 1:");
	scanf("%d",&m);
	printf("\nEnter column size of Matrix 1:");
	scanf("%d",&n);
	printf("\nEnter row size of Matrix 2:");
	scanf("%d",&p);
	printf("\nEnter column size of Matrix 2:");
	scanf("%d",&q);
	if(n==p)
	{
		printf("\nEnter Matrix 1 Elements:");
		for(i=0;i<m;i++)
		{
			for(j=0;j<n;j++)
			scanf("%d",&x[i][j]);
		}
		printf("\nEnter Matrix 2 Elements:");
		for(i=0;i<p;i++)
		{
			for(j=0;j<q;j++)
			scanf("%d",&y[i][j]);
		}
		for(i=0;i<m;i++)
		{
			for(j=0;j<q;j++)
			{
				z[i][j]=0;
				for(k=0;k<n;k++)
```

```
                                        z[i][j]+=x[i][k]*y[k][j];
                        }
                }
                printf("\nMultiplication Matrix Elements Are:");
                for(i=0;i<m;i++)
                {
                        printf("\n");
                        for(j=0;j<q;j++)
                        printf("%5d",z[i][j]);
                }
        }
        else
                printf("\nMATRIX MULTIPLICATION NOT POSSIBLE");
        getch( );
}
```

OUTPUT:

```
Enter row size of Matrix 1:2
Enter column size of Matrix 1:2
Enter row size of Matrix 2:2
Enter column size of Matrix 2:2
Enter Matrix 1 Elements:1 1 1 1
Enter Matrix 2 Elements:2 2 2 2
Multiplication Matrix Elements Are:
    4    4
    4    4
```

WRITE A C PROGRAM THAT USES FUNCTIONS TO PERFORM THE FOLLOWING:
I) ADDITION OF TWO MATRICES II) MULTIPLICATION OF TWO MATRICES

```
#include<stdio.h>
#include<conio.h>
int read_matrix(int a[10][10],int m,int n);

int write_matrix(int a[10][10],int m,int n);

void main( )

{

        int ch,i,j,m,n,p,q,k,r1,c1,a[10][10],b[10][10],c[10][10];

        clrscr( );

        printf("************************************");

        printf("\n\t\tMENU");

        printf("\n************************************");

        printf("\n[1]ADDITION OF TWO MATRICES");

        printf("\n[2]MULTIPLICATION OF TWO MATRICES");

        printf("\n[0]EXIT");

        printf("\n************************************");

        printf("\n\tEnter your choice:\n");

        scanf("%d",&ch);

        if(ch<=2&&ch>0)

            printf("Valid Choice\n");

        switch(ch)

        {

          case 1:       printf("Input rows and columns of A & B Matrix:");

                        scanf("%d%d",&r1,&c1);

                        printf("Enter elements of matrix A:\n");

                        read_matrix(a,r1,c1);

                        printf("Enter elements of matrix B:\n");

                        read_matrix(b,r1,c1);           /*Function call to read the matrix*/

                        printf("\n =====Matrix Addition=====\n");

                        for(i=0;i<r1;i++)

                        {

                                for(j=0;j<c1;j++)

                                c[i][j]=a[i][j]+b[i][j];

                        }
                        write_matrix(c,r1,c1);          /*Function call to write the matrix*/
                         break;
          case 2:       printf("Input rows and columns of A matrix:");
                        scanf("%d%d",&m,&n);
                        printf("Input rows and columns of B matrix:");
                        scanf("%d%d",&p,&q);
                        if(n= =p)
                        {
                             printf("matrices can be multiplied\n");
                             printf("resultant matrix is %d*%d\n",m,q);
                             printf("Input A matrix\n");
                             read_matrix(a,m,n);
                             printf("Input B matrix\n");
                             read_matrix(b,p,q);             /*Function call to read the matrix*/
                             printf("\n =====Matrix Multiplication=====\n");
                             for(i=0;i<m;++i)
                                  for(j=0;j<q;++j)
                                  {
                                        c[i][j]=0;
                                       for(k=0;k<n;++k)
                                            c[i][j]=c[i][j]+a[i][k]*b[k][j];
                                  }
                                  printf("Result of product two matrices:\n");
                                  write_matrix(c,m,q);   /*Function call to write the matrix*/
                        }    /*end if*/
                        else
                        {  printf("Matrices cannot be multiplied.");
                        }   /*end else*/
                        break;
            case 0:     printf("\n Choice Terminated");
                        break;
            default:    printf("\n Invalid Choice");
          }             /*switch*/
          getch ( );
}
```

```
/*Function read matrix*/
int read_matrix(int a[10][10],int m,int n)
{
   int i,j;
   for(i=0;i<m;i++)
           for(j=0;j<n;j++)
           scanf("%d",&a[i][j]);
   return 0;
}
/*Function to write the matrix*/
int write_matrix(int a[10][10],int m,int n)
{
   int i,j;
   for(i=0;i<m;i++)
   {
           for(j=0;j<n;j++)
           printf("%5d",a[i][j]);
           printf("\n");
   }
   return 0;
}
```

OUTPUT1:

```
************************************
            MENU
**********************************
[1]ADDITION OF TWO MATRICES
[2]MULTIPLICATION OF TWO MATRICES
[0]EXIT
**********************************
      Enter your choice:
0
 Choice Terminated
```

```
OUTPUT2:
************************************
          MENU
**********************************
[1]ADDITION OF TWO MATRICES
[2]MULTIPLICATION OF TWO MATRICES
[0]EXIT
**********************************
     Enter your choice:
1
Valid Choice
Input rows and columns of A & B Matrix:2
2
Enter elements of matrix A:
2
2
2
2
Enter elements of matrix B:
4
4
4
4

 =====Matrix Addition=====
   6   6
   6   6
```

```
OUTPUT3:
************************************
          MENU
**********************************
[1]ADDITION OF TWO MATRICES
[2]MULTIPLICATION OF TWO MATRICES
[0]EXIT
**********************************
     Enter your choice:
2
Valid Choice
Input rows and columns of A matrix:2
2
Input rows and columns of B matrix:2
2
matrices can be multiplied
resultant matrix is 2*2
Input A matrix
2
2
2
2
Input B matrix
2
2
2
2

 =====Matrix Multiplication=====
Result of product two matrices:
    8   8
    8   8
```

```
/*Write a program to insert an element into an array.*/
#include<stdio.h>
#include<conio.h>
void main( )
{
        int n,i,item,pos,x[10];
        clrscr( );
        printf("\nEnter how many elements:");
        scanf("%d",&n);
        printf("\nEnter %d Elements:",n);
        for(i=0;i<n;i++)
           scanf("%d",&x[i]);
        printf("\nEnter an Element to Insert:");
        scanf("%d",&item);
        printf("\nEnter Poistion to Insert:");
        scanf("%d",&pos);
        for(i=n-1;i>=pos-1;i--)
                x[i+1]=x[i];
        x[pos-1]=item;
        printf("\nArray Elements Are:");
        for(i=0;i<=n;i++)
                printf("%5d",x[i]);
        getch( );
}
OUTPUT:
Enter how many elements:5
Enter 5 Elements:1 2 3 4 5
Enter an Element to Insert:8
Enter Poistion to Insert:3
Array Elements Are:    1    2    8    3    4    5
```

```
/*Write a program to delete an element into an array.*/
#include<stdio.h>
#include<conio.h>
void main( )
{
        int n,i,item,pos,x[10];
        clrscr( );
        printf("\nEnter how many elements:");
        scanf("%d",&n);
        printf("\nEnter %d Elements:",n);
        for(i=0;i<n;i++)
           scanf("%d",&x[i]);
        printf("\nEnter Poistion to Delete:");
        scanf("%d",&pos);
        for(i=pos-1;i<=n-2;i++)
                x[i]=x[i+1];
        printf("\nArray Elements Are:");
        for(i=0;i<n-1;i++)
                printf("%5d",x[i]);
        getch( );
}
```

OUTPUT:

```
Enter how many elements:5

Enter 5 Elements:1 2 3 4 5

Enter Poistion to Delete:3

Array Elements Are:    1    2    4    5
```

/* PROGRAM FOR TIC-TAC-TOE GAME*/

```
#include<stdio.h>
#include<conio.h>
main( )
{
    int i;
    int player = 0; /* Player number - 1 or 2 */
    int winner = 0; /* The winning player */
    int choice = 0; /* Square selection number for turn */
    int row = 0; /* Row index for a square */
    int column = 0; /* Column index for a square */
    int line=0; /* Row or column index in checking loop */
    char board[3][3] = {                    /* The board */
                    {'1','2','3'}, /* Initial values are reference numbers */
                    {'4','5','6'}, /* used to select a vacant square for */
                    {'7','8','9'} /* a turn. */
                  };
/* The main game loop. The game continues for up to 9 turns */
/* As long as there is no winner */
    for(i=0;i<9&&winner==0;i++)
    {
    /* Display the board */
      printf("\n\n");
      printf(" %c | %c | %c\n", board[0][0], board[0][1], board[0][2]);
      printf("---+---+---\n");
      printf(" %c | %c | %c\n", board[1][0], board[1][1], board[1][2]);
      printf("---+---+---\n");
      printf(" %c | %c | %c\n", board[2][0], board[2][1], board[2][2]);
      player = i%2 + 1; /* Select player */
      /* Get valid player square selection */
      do
      {
        printf("\nPlayer %d, please enter the number of the square "
        "where you want to place your %c: ",
        player,(player==1)?'X':'O');
        scanf("%d", &choice);
        row = --choice/3; /* Get row index of square */
        column = choice%3; /* Get column index of square */
      }while(choice<0 || choice>9 || board[row][column]>'9');
      /* Insert player symbol */
       board[row][column] = (player == 1) ? 'X' : 'O';
       /* Check for a winning line . diagonals first */
       if((board[0][0]==board[1][1] && board[0][0]==board[2][2]) ||
       (board[0][2]==board[1][1] && board[0][2]==board[2][0]))
             winner = player;
       else
       /* Check rows and columns for a winning line */
       for(line = 0; line <= 2; line ++)
       if((board[line][0]==board[line][1] &&
       board[line][0]==board[line][2])||
       (board[0][line]==board[1][line] &&
       board[0][line]==board[2][line]))
       winner = player;
    }
/* Game is over so display the final board */
    printf("\n\n");
    printf(" %c | %c | %c\n", board[0][0], board[0][1], board[0][2]);
    printf("---+---+---\n");
    printf(" %c | %c | %c\n", board[1][0], board[1][1], board[1][2]);
    printf("---+---+---\n");
    printf(" %c | %c | %c\n", board[2][0], board[2][1], board[2][2]);
/* Display result message */
    if(winner == 0)
          printf("\nHow boring, it is a draw\n");
    else
          printf("\nCongratulations, player %d, YOU ARE THE
WINNER!\n",winner);
    getch( );
    return 0;
}
```

/* PROGRAM TO ARRAY ORDER REVERSAL*/

```
#include<stdio.h>
#include<conio.h>
#define size 100
void main( )
{
        int a[size],i,r,t,n;
        clrscr( );
        printf("\nEnter number of elements to enter into array");
        scanf("%d",&n);
        printf("\nEnter any %d numbers",n);
        for(i=0;i<n;i++)
        {
                scanf("%d",&a[i]);
        }
        printf("\nElements in array before reversal\n");
        for(i=0;i<n;i++)
                 printf("a[%d]=%d\t",i,a[i]);
        r=n/2;
        for(i=0;i<r;i++)
        {
                t=a[i];
                a[i]=a[n-1-i];
                a[n-1-i]=t;
        }
        printf("\nElements in array after reversal\n");
        for(i=0;i<n;i++)
                printf("a[%d]=%d\t",i,a[i]);
        getch( );
}
```

```
for(i=0,j=n-1;i<r;i++,j--)
    {
        a[i]=a[i]+a[j];
        a[j]=a[i]-a[j];
        a[i]=a[i]-a[j];
    }
```

OUTPUT:

```
Enter number of elements to enter into array6

Enter any 6 numbers   1        2        3        4        5        6

Elements in array before reversal
a[0]=1  a[1]=2  a[2]=3  a[3]=4  a[4]=5  a[5]=6
Elements in array after reversal
a[0]=6  a[1]=5  a[2]=4  a[3]=3  a[4]=2  a[5]=1
```

/* PROGRAM TO REMOVE DUPLICATES FROM AN ORDERED ARRAY*/

```
#include<stdio.h>
#include<conio.h>
void main( )
{
        int a[20],i,j,k,n;
        clrscr( );
        printf("\nEnter array size : ");
        scanf("%d",&n);
        printf("\nAccept Numbers :\n");
        for(i=0;i<n;i++)
                scanf("%d",&a[i]);
        printf("\nOriginal array is:\n");
        for(i=0;i<n;i++)
                printf("%d",a[i]);
        printf("\nUpdated array is:\n");
        for(i=0;i<n;i++)
        {
                for(j=i+1;j<n;)
                {
                        if(a[j]==a[i])
                        {
                                for(k=j;k<n;k++)
                                        a[k]=a[k+1];
                                n--;
                        }
                        else
                                j++;
                }
        }
        for(i=0;i<n;i++)
                printf("%d ",a[i]);
        getch( );
}
```

OUTPUT:

```
Enter array size : 8

Accept Numbers :
1 2 2 3 4 4 4 5

Original array is:
1 2 2 3 4 4 4 5
Updated array is:
1 2 3 4 5
```

STRING BASICS

A string is an array of characters. A **string constant** is a sequence of characters or symbols between a pair of double-quote characters. Anything between a pair of double quotes is interpreted by the compiler as a string, including any special characters and embedded.

The end of the string is marked with a special character, the ***null character***, which is simply the character with the value 0. (The null character has no relation except in name to the *null pointer*. In the ASCII character set, the null character is named NULL.) The null or string-terminating character is represented by another character escape sequence, **0**.

The null character is not counted towards the length of the string, but takes memory of one element. Null character cannot be read or written, but is used internally only.

Any constant string is defined between double quotation marks. Whenever we write a string, enclosed in double quotes, C automatically creates an array of characters for us, containing that string, terminated by the **\0** character. For example, we can declare and define an array of characters, and initialize it with a string constant:

char string[] = "Hello, world!";

All the string handling functions are prototyped in: string.h or stdio.h standard header file. So while using any string related function, don't forget to include either **stdio.h** or **string.h**.

Declaration and Initialization string variables:

A string variable is any valid C variable name and is always declared as an array.

General Format: **char variable_name[size];**

The size determines the number of characters in string name.

Ex: **char city [25];** **char name [50];**

When a compiler assigns a character string to a character array, it automatically supplies a NULL character '\0' at the end of the string. Therefore the size should be equal to maximum number of characters in the string plus one.

Character array may be initializes when they are declared.

char name [10] = "hello" **or**

char name[10]={'h','e','l','l','o','\0'}

Arrays of Strings

We can use a two-dimensional array of elements of type char to store strings, where each row is used to hold a separate string. In this way we could arrange to store a whole bunch of strings and refer to any of them through a single variable name;

EG:
```
char sayings [3][32] = {
                    "Manners maketh man." ,
                    "Many hands make light work.”,
                    "Too many cooks spoil the broth."
                };
```

This creates an array of three rows of 32 characters. The strings between the braces will be assigned in sequence to the three rows of the array, sayings [0], sayings [1], and sayings [2]. Note that you don’t need braces around each string. The compiler can deduce that each string is intended to initialize one row of the array. The last dimension is specified to be 32, which is just sufficient to accommodate the longest string, including its terminating \0 character. The first dimension specifies the number of strings.

When you’re referring to an element of the array—sayings[i][j], for instance—the first index, i, identifies a row in the array, and the second index, j, identifies a character within a row. When you want to refer to a complete row containing one of the strings, you just use a single index value between square brackets. For instance, sayings [1] refers to the second string in the array, "Many hands make light work.".

Although you must specify the last dimension in an array of strings, you can leave it to the compiler to figure out how many strings there are:

```
char sayings[ ][32] = {
                "Manners maketh man.",
                "Many hands make light work.",
                "Too many cooks spoil the broth."
            };
```

If the first dimension in the array is omitted here, so the compiler will deduce this from the initializers between braces. Because we have three initializing strings, the compiler will make the first array dimension 3. Of course, you must still make sure that the last dimension is large enough to accommodate the longest string, including its terminating null character.

Inputting Strings Using the scanf() Function

The input function scanf ( ) can be used with %s format specification to read a string. When we call scanf with a string variable as an argument, we must remember that array output arguments are always passed to functions by sending the address of the initial array elements. Therefore we do not apply the address-operator to a string argument passed to scanf or to any other function.

EX: **char city[20];**

scanf("%s",city);

To read a string, include the specifier **%s** in scanf( )'s format string. How does scanf( ) decide where the string begins and ends? The beginning is the first nonwhitespace character encountered. The end can be specified in one of two ways. If you use %s in the format string, the string runs up to (but not including) the next whitespace character (space, tab, or newline). If you use %ns (where n is an integer constant that specifies field width), scanf( ) inputs the next n characters or up to the next whitespace character, whichever comes first.

You can read in multiple strings with scanf( ) by including more than one %s in the format string. For example: **scanf("%s%s%s", s1, s2, s3);**

If in response to this statement you enter January February March, January is assigned to the string s1, February is assigned to s2, and March to s3.

What about using the field-width specifier? If you execute the statement

scanf("%3s%3s%3s", s1, s2, s3); and in response you enter September, Sep is assigned to s1, tem is assigned to s2, and ber is assigned to s3.

What if you enter fewer or more strings than the scanf( ) function expects?

If you enter fewer strings, scanf( ) continues to look for the missing strings, and the program doesn't continue until they're entered. For example, if in response to the statement

scanf("%s%s%s", s1, s2, s3); you enter January February, the program waits for the third string specified in the scanf( ) format string. If you enter more strings than requested, the unmatched strings remain pending (waiting in the keyboard buffer) and are read by any subsequent scanf( ) or other input statements. For example, if in response to the statements

scanf("%s%s", s1, s2); scanf("%s", s3); you enter January February March, the result is that January is assigned to the string s1, February is assigned to s2, and March is assigned to s3.

The scanf( ) function has a return value, an integer value equaling the number of items successfully inputted. The return value is often ignored. When you're reading text only, the gets( ) function is usually preferable to scanf( ). It's best to use the scanf( ) function when you're reading in a combination of text and numeric data.

Inputting Strings Using the gets( ) Function

The gets( ) function gets a string from the keyboard. When gets( ) is called, it reads all characters typed at the keyboard up to the first newline character (which you generate by pressing Enter). This function discards the newline, adds a null character, and gives the string to the calling program. The string is stored at the location indicated by a pointer to type char passed to gets( )

Printing a String:

The print function printf ( ) can be used with %s format specification to print a string. When printf( ) encounters a %s in its format string, the function matches the %s with the corresponding argument in its argument list. For a string, this argument must be a pointer to the string that you want displayed.

The printf( ) function displays the string on-screen, stopping when it reaches the string's terminating null character. For example:

char *str = "A message to display";

printf("%s", str);

You can also display multiple strings and mix them with literal text and/or numeric variables:

char *bank = "obc";

char *name = "knreddy";

int balance = 10000;

printf("The balance at %s for %s is %d.", bank, name, balance);

The resulting output is **The balance at obc for knreddy is 10000.**

The puts( ) Function

The puts( ) function puts a string on-screen--hence its name. A pointer to the string to be displayed is the only argument puts( ) takes. Because a literal string evaluates as a pointer to a string, puts( ) can be used to display literal strings as well as string variables. The puts( ) function automatically inserts a newline character at the end of each string it displays, so each subsequent string displayed with puts( ) is on its own line.

```
/* write a program to read and write string using scanf( ) and printf( ) */
#include<stdio.h>
#include<conio.h>
#define SIZE 50
 void main( )
{
        char str[SIZE];
        clrscr( );
        printf("\nEnter a string ");
        scanf("%s",str);
        printf("\nEntered string is ");
        printf("%s",str);
        getch( );
}
```

OUTPUT:

```
Enter a string knreddy
Entered string is knreddy
```

```
/* write a program to read and write string using gets( ) and puts( ) */

#include<stdio.h>

#include<conio.h>

#define SIZE 50

void main( )

{

        char str[SIZE];

        clrscr( );

        printf("\nEnter a string ");

        gets(str);

        printf("\nEntered string is ");

        puts(str);

        getch( );

}
```

OUTPUT:

```
Enter a string knreddy



Entered string is knreddy
```

We can also read the string character by character, which is similar to reading the elements of an array of integers with a difference that the end of input is detected by press of Enter key.

```
#include<stdio.h>
#include<conio.h>
#define SIZE 50
void main( )
{
        char c,str[SIZE];
        int i=0,j=0;
        clrscr( );
        printf("enter a string char by char :");
        while((c=getchar())!='\n')
        {
            str[i]=c;
            i++;
        }
        str[i]='\0';
        printf("The characters entered are :");
        while(str[j]!='\0')
        {
             printf("%c",str[j]);
             j++;
        }
         getch( );
}
OUTPUT:
enter a string char by char :KNREDDY
The characters entered are :KNREDDY
```

PROCESSING STRINGS

Strings can be processed either by using some predefined functions or by processing all characters individually.

/* WRITE A PROGRAM TO READ & WRITE STRING USING scanf( ) AND printf( ) */

```
#include<stdio.h>
#include<conio.h>
#define SIZE 50
 void main( )
{
            char str[SIZE];
            clrscr( );
            printf("\nEnter a string ");
            scanf("%s",str);
            printf("\nEntered string is ");
            printf("%s",str);
            getch( );
}
```

```
OUTPUT:
Enter a string knreddy
Entered string is knreddy
```

/* WRITE A PROGRAM TO READ AND WRITE STRING USING gets( ) AND puts( ) */

```
#include<stdio.h>
#include<conio.h>
#define SIZE 50
void main( )
{
            char str[SIZE];
            clrscr( );
            printf("\nEnter a string ");
            gets(str);
            printf("\nEntered string is ");
            puts(str);
            getch( );
}
```

```
OUTPUT:
Enter a string knreddy
Entered string is knreddy
```

We can also read the string character by character, which is similar to reading the elements of an array of integers with a difference that the end of input is detected by press of Enter key.

```
#include<stdio.h>
#include<conio.h>
#define SIZE 50
void main( )
{
                char c,str[SIZE];
                int i=0,j=0;
                clrscr( );
                printf("enter a string char by char :");
                while((c=getchar())!='\n')
                {
                        str[i]=c;
                        i++;
                }
                str[i]='\0';
                printf("The characters entered are :");
                while(str[j]!='\0')
                {
                         printf("%c",str[j]);
                         j++;
                }
                 getch( );
}
```

OUTPUT:

```
enter a string char by char :KNREDDY
The characters entered are :KNREDDY
```

CHARACTER ARITHMETIC

/* PROGRAM TO CONVERT LOWER CASE TEXT TO UPPER CASE*/

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
void main( )
{
      char text[100];
      int i,len;

      printf("\nEnter aline of text\n");
      scanf("%[^\n]",text);
      printf("\nThe text entered is\n");
      puts(text);
      len=strlen(text);
      for(i=0;i<len;i++)
      {
          if((text[i]>='a')&&(text[i]<='z'))
              text[i]=text[i]+'A'-'a';
       }
       printf("\nThe text in upper case is\n");
       puts(text);
       getch( );
}
```

OUTPUT:

```
Enter aline of text
this is C prog

The text entered is
this is C prog

The text in upper case is
THIS IS C PROG
```

/* PROGRAM TO FIND STRING LENGTH AND COPY STRINGS USING USERDEFINED FUNCTIONS*/

```
#include<stdio.h>
#include<conio.h>
int strlength(char str[ ])
{
        int count=0;
        while(str[count]!='\0')
                count++;
        return(count);
}
void strcopy(char dest[],char src[])
{
        int i,length;
        length=strlength(src);
        for(i=0;i<length;i++)
                dest[i]=src[i];
        dest[length]='\0';
}
void main( )
{
        int len;
        char str1[100],str2[100];
        clrscr( );
        printf("\nEnter a string :");
        gets(str1);
        len=strlength(str1);
        printf("\nThe length of string is %d",len);
        strcopy(str2,str1);
        printf("\nSource string is %s\nCopied string is %s",str1,str2);
        getch( );
}
```

OUTPUT:

```
Enter a string :c program
The length of string is 9
Source string is c program
Copied string is c program
```

/*PROGRAM TO COMPARE TWO STRINGS USING USER DEFINED PROGRAMS */

```
#include<stdio.h>
#include<conio.h>
int strlength(char str[])
{
   int count=0;
   while(str[count]!='\0')
        count++;
   return(count);
}
int strcompare(char str1[],char str2[])
{
   int i,length,length1,length2;
   length1=strlength(str1);
   length2=strlength(str2);
   length=(length1<length2)?length1:length2;
   for(i=0;i<=length;i++)
       if(str1[i]<str2[i])
           return -1;
       else if(str1[i]>str2[i])
            return 1;
   return 0;
}
void main( )
{
     int status;
     char str1[100],str2[100];
     clrscr( );
     printf("\nEnter first string");
     gets(str1);
     printf("\nEnter second string");
     gets(str2);
     status=strcompare(str1,str2);
     if(status==-1)
        printf("\nFirst string is less than second string");
```

```
    else if(status==1)
      printf("\nFirst string is greater than second string");
    else
      printf("\nBoth strings are equal");
    getch( );
}
```

OUTPUT:

```
Enter first string cprog

Enter second string Cprogram

First string is greater than second string
```

OUTPUT:

```
Enter first string KNR

Enter second string KNR

Both strings are equal
```

STRING LIBRARY FUNCTIONS

Text data, which C stores in strings, is an important part of many programs. C offers a variety of functions for other types of string manipulation as well.

- How to determine the length of a string
- How to copy and join strings
- About functions that compare strings
- How to search strings
- How to convert strings
- How to test characters

STRING LENGTH

In C programs, a string is a sequence of characters, with its beginning indicated by a pointer and its end marked by the null character \0. At times, you need to know the length of a string (the number of characters between the start and the end of the string). This length is obtained with the library function **strlen( )**. Its prototype, in **STRING.H**, is size_t strlen(char *str);

size_t is defined in STRING.H as unsigned, so the function strlen( ) returns an unsigned integer. The size_t type is used with many of the string functions.

The argument passed to strlen is a pointer to the string of which you want to know the length. The function strlen( ) returns the number of characters between str and the next null character, not counting the null character.

***syntax:* strlen (str);**

Program to demonstrate string length

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
void main( )
{
    char str[100];
   clrscr( );
    printf("enter any string: ");
    scanf("%s",str);
    printf("length of the string is %d :",strlen(str));
    getch( );
}
OUTPUT
enter any string: knreddy
length of the string is : 7
```

COPYING STRINGS

The C library has three functions for copying strings. Because of the way C handles strings, you can't simply assign one string to another, as you can in some other computer languages. You must copy the source string from its location in memory to the memory location of the destination string. The string-copying functions are **strcpy( ), strncpy( ),** and **strdup( ).** All of the string-copying functions require the header file STRING.H.

The strcpy( ) Function

The library function strcpy( ) copies an entire string to another memory location. Its prototype is as follows: **char *strcpy(char *destination, char *source);**

The function strcpy( ) copies the string (including the terminating null character \0) pointed to by source to the location pointed to by destination. The return value is a pointer to the new string, destination. When using strcpy( ), you must first allocate storage space for the destination string. The function has no way of knowing whether destination points to allocated space. If space hasn't been allocated, the function overwrites strlen(source) bytes of memory, starting at destination; this can cause unpredictable problems.

Example program to demonstrate strcpy( )

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
char source[ ] = "The source string.";
void main( )
{
    char dest1[80];
    char *dest2, *dest3;
    printf("\nsource: %s", source );
    /* Copy to dest1 is okay because dest1 points to */
    /* 80 bytes of allocated space. */
    strcpy(dest1, source);
    printf("\ndest1: %s", dest1);
    /* To copy to dest2 you must allocate space. */
    dest2 = (char *)malloc(strlen(source) +1);
    strcpy(dest2, source);
    printf("\ndest2: %s\n", dest2);
    /* Copying without allocating destination space is a no-no. */
    /* The following could cause serious problems. */
    /* strcpy(dest3, source); */
    getch( );
}
OUTPUT:
source: The source string.
dest1: The source string.
dest2: The source string.
```

The strncpy( ) Function

The strncpy( ) function is similar to strcpy( ), except that strncpy( ) lets you specify how many characters to copy. Its prototype is **char *strncpy(char *destination, char *source, size_t n);**

The arguments destination and source are pointers to the destination and source strings. The function copies, at most, the first n characters of source to destination. If source is shorter than n characters, enough null characters are added to the end of source to make a total of n characters copied to destination. If source is longer than n characters, no terminating \0 is added to destination. The function's return value is destination.

Eg program to demonstrate strncpy( )

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
char source[ ] = "The source string.";
void main( )
{
    char dest1[80];
    int n;
    clrscr( );
    printf("\nsource: %s", source );
    printf("\nEnter number of characters to copy from source: ");
    scanf("%d",&n);
    strncpy(dest1, source,n);
    printf("\ndest1: %s", dest1);
    getch( );
}
```

```
OUTPUT:
source: The source string.
Enter number of characters to copy from source: 5
dest1: The s
--------------------------------------------
OUTPUT:
source: The source string.
Enter number of characters to copy from source: 20
dest1: The source string.
```

The strdup( ) Function

The library function strdup( ) is similar to strcpy(), except that strdup( ) performs its own memory allocation for the destination string with a call to malloc( ). In effect, it does malloc( ) and then calling strcpy( ). The prototype for strdup( ) is **char *strdup(char *source);**

The argument source is a pointer to the source string. The function returns a pointer to the destination string--the space allocated by malloc( )--or NULL if the needed memory couldn't be allocated.. Note that strdup( ) isn't an ANSI-standard function. It is included in the Microsoft, Borland, and Symantec C libraries, but it might not be present (or it might be different) in other C compilers.

CONCATENATING STRINGS

Concatenation means to join two strings-to append one string onto the end of another. The C standard library contains two string concatenation functions **strcat( )** and **strncat( ) ;** both of which require the header file **STRING.H.**

The strcat( ) Function :

The prototype of strcat( ) is **char *strcat(char *str1, char *str2);**

The function appends a copy of str2 onto the end of str1, moving the terminating null character to the end of the new string. You must allocate enough space for str1 to hold the resulting string. The return value of strcat( ) is a pointer to str1.

Ex1 program using strcat( ) to concatenate strings

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
void main( )
{
    char str1[20],str2[20];
    clrscr( );
    printf("enter string1 ");
    scanf("%s",str1);
    printf("enter string2 ");
    scanf("%s",str2);
    printf("concatenation of string2 to end of sting1 is:");
    strcat(str1,str2);
    puts(str1);
    getch( );
  }
OUTPUT
enter string1 good
enter string2 students
concatenation of string2 to end of sting1 is:goodstudents
```

Ex2 program using strcat( ) to concatenate strings

```
/* The strcat( ) function. */

#include <stdio.h>

#include<conio.h>

#include <string.h>

char str1[27] = "a";

char str2[2];

void main( )

{

      int n;

     clrscr( );

      /* Put a null character at the end of str2[]. */

      str2[1] ='\0';

      for (n =98; n<123; n++)

      {

          str2[0] =n;

          strcat(str1, str2);

          puts(str1);

      }

 getch( );

}
```

OUTPUT:

```
ab
abc
abcd
abcde
abcdef
abcdefg
abcdefgh
abcdefghi
abcdefghij
abcdefghijk
abcdefghijkl
abcdefghijklm
abcdefghijklmn
abcdefghijklmno
abcdefghijklmnop
abcdefghijklmnopq
abcdefghijklmnopqr
abcdefghijklmnopqrs
abcdefghijklmnopqrst
abcdefghijklmnopqrstu
abcdefghijklmnopqrstuv
abcdefghijklmnopqrstuvw
abcdefghijklmnopqrstuvwx
abcdefghijklmnopqrstuvwxy
abcdefghijklmnopqrstuvwxyz
```

The strncat( ) Function

The library function strncat( ) also performs string concatenation, but it lets you specify how many characters of the source string are appended to the end of the destination string. The prototype is

char *strncat(char *str1, char *str2, size_t n);

If str2 contains more than n characters, the first n characters are appended to the end of str1. If str2 contains fewer than n characters, all of str2 is appended to the end of str1. In either case, a terminating null character is added at the end of the resulting string. You must allocate enough space for str1 to hold the resulting string. The function returns a pointer to str1.

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
void main( )
{
    char str1[20],str2[20];
    int n;
    clrscr( );
    printf("enter string1: ");
    scanf("%s",str1);
    printf("enter string2: ");
    scanf("%s",str2);
    printf("enter any how many characters are to be append ");
    scanf("%d",&n);
    printf("concatenation of %d characetrs from string2 to end of sting1 is:",n);
    strncat(str1,str2,n);
    puts(str1);
    getch( );
 }
OUTPUT1:
enter string1: computer
enter string2 : language
enter any how many characters are to be append 5
concatenation of 5 characetrs from string2 to end of sting1 is:computerlangu
OUTPUT2:
enter string1: c
enter string2: programming
enter any how many characters are to be append 5
concatenation of 5 characetrs from string2 to end of sting1 is:cprogr
```

Ex prog using the strncat( ) function to concatenate strings.

```
/* The strncat( ) function. */

#include<stdio.h>

#include<conio.h>

#include<string.h>

char str2[] = "abcdefghijklmnopqrstuvwxyz";

void main( )

{

    char str1[27];

    int n;

    clrscr ( );

    for (n=1; n< 27; n++)

    {

        str1[0]=" ";

        strncat(str1, str2, n);

        puts(str1);

    }

    getch( );

}
```

```
OUTPUT:
a
ab
abc
abcd
abcde
abcdef
abcdefg
abcdefgh
abcdefghi
abcdefghij
abcdefghijk
abcdefghijkl
abcdefghijklm
abcdefghijklmn
abcdefghijklmno
abcdefghijklmnop
abcdefghijklmnopq
abcdefghijklmnopqr
abcdefghijklmnopqrs
abcdefghijklmnopqrst
abcdefghijklmnopqrstu
abcdefghijklmnopqrstuv
abcdefghijklmnopqrstuvw
abcdefghijklmnopqrstuvwx
abcdefghijklmnopqrstuvwxy
abcdefghijklmnopqrstuvwxyz
```

COMPARING STRINGS

Strings are compared to determine whether they are equal or unequal. If they are unequal, one string is "greater than" or "less than" the other. Determinations of "greater" and "less" are made with the ASCII codes of the characters. In the case of letters, this is equivalent to alphabetical order, with the one seemingly strange exception that all uppercase letters are "less than" the lowercase letters. This is true because the uppercase letters have ASCII codes 65 through 90 for A through Z, while lowercase a through z are represented by 97 through 122. Thus, "ZEBRA" would be considered to be less than "apple" by these C functions.

The ANSI C library contains functions for two types of string comparisons: comparing two entire strings, and comparing a certain number of characters in two strings.

Comparing Two Entire Strings

The function strcmp( ) compares two strings character by character. Its prototype is :

int strcmp(char *str1, char *str2);

The arguments str1 and str2 are pointers to the strings being compared. The function's return values are given in following table**.**

Return Value	**Meaning**
< 0	str1 is less than str2
0	str1 is equal to str2.
> 0	str1 is greater than str2.

1. If two strings are equal, the return value is zero. Two strings are considered equal if they are the same length and all characters in the same relative positions are equal.

2. If the first parameter is less than the second parameter, the return value is less than zero. A string, s1, is less than another string, s2, if starting from the first character; we can find a character, in s1 that is less than the character in s2. Note that when the end of either string is reached, the NULL character is compared with the corresponding character in the other string.

3. If the first parameter is greater than the second parameter, the return value is greater than zero. A string, s1, is greater than another string, s2, if starting from the first character; we can find a character, in s1 that is greater than the corresponding character in s2. Note that when the end of either string is reached, the NULL character is compared with the corresponding character in the other string.

Ex prog to demonstrate strcmp( );

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
void main( )
{
    char str1[20],str2[20];
    int i;
    printf("enter string 1:");
    gets(str1);
    printf("enter string 2:");
    gets(str2);
    i=strcmp(str1,str2);
    printf("the return value of strcmp is i=%d",i);
    if (i==0)
            printf("\nTwo strings are equal");
    else
             printf("\nTwo strings are not equal");
    getch( );
 }
```

OUTPUT 1:

```
enter string 1:knreddy
enter string 2:knreddy
the return value of strcmp is i=0
Two strings are equal
```

OUTPUT 2:

```
enter string 1:abcd
enter string 2:efgh
the return value of strcmp is i=-1
Two strings are not equal
```

OUTPUT 3:

```
enter string 1: efgh
enter string 2: abcd
the return value of strcmp is i=1
Two strings are not equal
```

OUTPUT 4:

```
enter string 1:KNR
enter string 2:knr
the return value of strcmpi is i=-1
Two strings are not equal
```

Ex2:

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
void main( )
{
    char s[80];        int i;
    clrscr( );
    printf("Enter password : ");
    gets(s);
    i = strcmp(s,"cpds");
    if (i==0)
            printf("\nCorrect password.");
    else
             printf("\nIncorrect password.");
    getch( );
}
```

OUTPUT 1:

```
Enter password : cpds
Correct password.
```

OUTPUT 2:

```
Enter password : knreddy
Incorrect password.
```

OUTPUT 3:

```
Enter password : cpd
Incorrect password.
```

EXPLANATION: In the above program password is **cpds.** Whenever the user enters password; it is compared with the original password. If both are same it returns 0.

Comparing Partial Strings

The library function strncmp( ) compares a specified number of characters of one string to another string. Its prototype is **int strncmp(char *str1, char *str2, size_t n);**

The function strncmp( ) compares n characters of str2 to str1. The comparison proceeds until n characters have been compared or the end of str1 has been reached. The method of comparison and return values are the same as for strcmp( ). The comparison is case-sensitive.

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
void main( )
{
    char str1[20],str2[20];    int i,n;
    clrscr( );
    printf("enter string 1:");
    gets(str1);
    printf("enter string 2:");
    gets(str2);
    printf("how many characters of str2 are compared with str1 ");
    scanf("%d",&n);
    i=strncmp(str1,str2,n);
    printf("the return value of strncmp is i=%d",i);
    getch( );
}
```

```
OUTPUT 1 :
enter string 1:knreddy
enter string 2:knr
how many characters of str2 are compared with str1 3
the return value of strncmp is i=0

OUTPUT 2 :
enter string 1:knreddy
enter string 2:knr
how many characters of str2 are compared with str1 4
the return value of strncmp is i=101
```

Comparing Two Strings While Ignoring Case

Unfortunately, the ANSI C library doesn't include any functions for case-insensitive string comparison. Fortunately, most C compilers provide their own "in-house" functions for this task. Microsoft uses a function called stricmp( ). Borland has two functions--strcmpi( ) and stricmp( ). You need to check your library reference manual to determine which function is appropriate for your compiler.When you use a function that isn't case-sensitive, the strings “KNREDDY” and “knreddy” compare as equal.

Ex: str1=”KNREDDY”

str2=”knreddy”

i=strcmpi(str1,str2) Here the return value is zero. The function strcmpi ignores case of characters.

SEARCHING STRINGS

The C library contains a number of functions that search strings. To put it another way, these functions determine whether one string occurs within another string and, if so, where. There are six string-searching functions, all of which require the header file STRING.H.

The strchr( ) Function

The strchr( ) function finds the first occurrence of a specified character in a string. The prototype is **char *strchr(char *str, int ch);**

The function strchr( ) searches str from left to right until the character ch is found or the terminating null character is found. If ch is found, a pointer to it is returned. If not, NULL is returned. When strchr( ) finds the character, it returns a pointer to that character. Knowing that str is a pointer to the first character in the string, you can obtain the position of the found character by subtracting str from the pointer value returned by strchr( ). Remember that the first character in a string is at position 0. Like many of C's string functions, strchr( ) is case-sensitive. For example, it would report that the character F isn't found in the string raffle.

Eg:

```
#include<stdio.h>

#include<conio.h>

#include<string.h>

void main( )

{

        char str[20],s,*ch;

         int i;

        clrscr( );

        printf("\nenter a string :");

        gets(str);

        printf("\nenter a character to search in the string:");
        s=getchar();
        ch=strchr(str,s);
        if(ch=='\0')
        {
                printf("character %c is not found in string %s",s,str);
        }
        else
        {
                i=ch-str;
                printf("first occurence of character %c in string %s is at the position %d",s,str,i+1);
        }
        getch( );
   }
```

OUTPUT 1:

```
enter a string :knreddy
enter a character to search in the string:d
first occurence of character d in string knreddy is at the position 5
```

OUTPUT 2:

```
enter a string :knreddy
enter a character to search in the string:x
character x is not found in string knreddy
```

The strrchr( ) Function

The library function strrchr( ) is identical to strchr( ), except that it searches a string for the last occurrence of a specified character in a string. Its prototype is **char *strrchr(char *str, int ch);** The function strrchr( ) returns a pointer to the last occurrence of ch in str and NULL if it finds no match.

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
void main( )
{
        char str[20],s,*ch;
        int i;
        clrscr( );
        printf("\nenter a string :");
        gets(str);
        printf("\nenter a character to search in the string:");
        s=getchar();
        ch=strrchr(str,s);
        if(ch=='\0')
        {
                printf("character %c is not found in string %s",s,str);
        }
        else
        {
                i=ch-str;
                printf("last occurence of character %c in string %s is at the position %d",s,str,i+1);
        }
        getch( );
 }
OUTPUT:
enter a string :knreddy
enter a character to search in the string:d
last occurence of character d in string knreddy is at the position 6
```

The strcspn( ) Function

The library function strcspn( ) searches one string for the first occurrence of any of the characters in a second string. Its prototype is **size_t strcspn(char *str1, char *str2);**

The function strcspn( ) starts searching at the first character of str1, looking for any of the individual characters contained in str2. This is important to remember. The function doesn't look for the string str2, but only the characters it contains. If the function finds a match, it returns the offset from the beginning of str1, where the matching character is located. If it finds no match, strcspn( ) returns the value of strlen(str1). This indicates that the first match was the null character terminating the string.

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
void main( )
{
    char str1[20],str2[20],s;      int i;
   clrscr( );
    printf("\nenter a string1 :");
    gets(str1);
    printf("\nenter a string2 :");
    gets(str2);
    i=strcspn(str1,str2);
    if(i==strlen(str1))
    {
           printf("any character of %s is not found in %s",str1,str2);
    }
    else
    {
          printf("character %c in string %s is at the position %d in string %s",str1[i],str2,i+1,str1);
    }
    getch( );
   }
OUTPUT:
enter a string1 :knreddy
enter a string2 :adr
character r in string adr is at the position 3 in string knreddy
```

The strspn( ) Function

This function is related to the previous one, strcspn( ).

Its prototype is **size_t strspn(char *str1,char *str2);**

The function strspn( ) searches str1, comparing it character by character with the characters contained in str2. It returns the position of the first character in str1 that doesn't match a character in str2. In other words, strspn( ) returns the length of the initial segment of str1 that consists entirely of characters found in str2. The return is 0 if no characters match.

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
void main( )
{
    char str1[20],str2[20],s;       int i;
    clrscr( );
    printf("\nenter a string1 :");
    gets(str1);
    printf("\nenter a string2 :");
    gets(str2);
    i=strspn(str1,str2);
    if(i==0)
    {
        printf("the first character of string %s and string %s are not same ",str2,str1);
    }
    else
    {
        printf("the first %d characters of string %s and string %s are same ",i,str1,str2);
    }
    getch( );
}
```

OUTPUT 1:	OUTPUT 2:
enter a string1 :knreddy enter a string2 :knr the first 3 characters of string knreddy and string knr are same	enter a string1 :knreddy enter a string2 :cpds the first character of string cpds and string knreddy are not same

The strpbrk( ) Function

The library function strpbrk( ) is similar to strcspn( ), searching one string for the first occurrence of any character contained in another string. It differs in that it doesn't include the terminating null characters in the search. The function prototype is **char *strpbrk(char *str1, char *str2);**

The function strpbrk( ) returns a pointer to the first character in str1 that matches any of the characters in str2. If it doesn't find a match, the function returns NULL. You can obtain the offset of the first match in str1 by subtracting the pointer str1 from the pointer returned by strpbrk( ) (if it isn't NULL, of course).

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
void main( )
{
    char str1[20],str2[20],s,*ch;       int i;
    clrscr( );
    printf("\nenter a string1 :");
    gets(str1);
    printf("\nenter a string2 :");
    gets(str2);
    ch=strpbrk(str1,str2);
    if(ch=='\0')
            printf("any character of %s is not found in %s",str2,str1);
    else
    {
        i=ch-str1;
        printf("character %c in string %s is at the position %d in string %s",str1[i],str2,i+1,str1);
    }
    getch( );
}
```

```
OUTPUT:
enter a string1 :knreddy
enter a string2 :red
character r in string red is at the position 3 in
string knreddy
```

```
OUTPUT:
enter a string1 :cpds
enter a string2 :eee
any character of eee is not found in cpds
```

The strstr( ) Function

The final, and perhaps most useful, C string-searching function is strstr( ). This function searches for the first occurrence of one string within another, and it searches for the entire string, not for individual characters within the string. Its prototype is **char *strstr(char *str1, char *str2);**

The function strstr( ) returns a pointer to the first occurrence of str2 within str1. If it finds no match, the function returns NULL. If the length of str2 is 0, the function returns str1. When strstr() finds a match, you can obtain the offset of str2 within str1 by pointer subtraction, as explained earlier for strchr( ). The matching procedure that strstr( ) uses is case-sensitive.

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
void main( )
{
    char str1[20],str2[20],s,*ch;   int i;
    clrscr( );
    printf("\nenter a string1 :");
    gets(str1);
    printf("\nenter a string2 :");
    gets(str2);
    ch=strstr(str1,str2);
    if(ch=='\0')
      printf("\nstring %s is not found in string %s",str2,str1);
    else
    {
     i=ch-str1;
     printf("\nstring %s is sub-string of string %s",str2,str1);
     printf("\nstring %s start at the position %d in string %s",str2,i+1,str1);
    }
    getch( );
 }
```

```
OUTPUT:
enter a string1 :knreddycpds
enter a string2 :cpd
string cpd is sub-string of string knreddycpds
string cpd start at the position 8 in string
knreddycpds
```

```
OUTPUT:
enter a string1 :knreddy
enter a string2 :cpds
string cpds is not found in string knreddy
```

STRING CONVERSIONS

Many C libraries contain two functions that change the case of characters within a string.

Their prototypes, in STRING.H, are as follows:

char *strlwr(char *str);

char *strupr(char *str);

The function strlwr() converts all the letter characters in str from uppercase to lowercase; strupr( ) does the reverse, converting all the characters in str to uppercase. Nonletter characters aren't affected. Both functions return str. Note that neither function actually creates a new string but modifies the existing string in place.

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
void main( )
{
    char str1[20],str2[20];
    printf("\nenter string1 in upper-case :");
    gets(str1);
    printf("\nenter string2 in lower-case :");
    gets(str2);
    strlwr(str1);
    puts(str1);
    strupr(str2);
    puts(str2);
    getch( );
}
```

OUTPUT:

```
enter string1 in upper-case :KNR
enter string2 in lower-case :cpds
knr
CPDS
```

OUTPUT:

```
enter string1 in upper-case :CPROGRAMMING
enter string2 in lower-case :clab3
cprogramming
CLAB3
```

MISCELLANEOUS STRING FUNCTIONS

There are few string functions that don't fall into any other category. They all require the header file

STRING.H.

The strrev( ) Function

The function strrev( ) reverses the order of all the characters in a string. Its prototype is

char *strrev(char *str);

The order of all characters in str is reversed, with the terminating null character remaining at the end. The function returns str.

The strset( ) and strnset( ) Functions

Like the previous function, strrev( ), strset( ) and strnset( ) aren't part of the ANSI C standard library. These

functions change all characters (strset( )) or a specified number of characters (strnset( )) in a string to a specified character. The prototypes are

char *strset(char *str, int ch);

char *strnset(char *str, int ch, size_t n);

The function strset( ) changes all the characters in str to ch except the terminating null character. The function strnset( ) changes the first n characters of str to ch. If n >= strlen(str), strnset( ) changes all the characters in str.

```
/* Demonstrates strrev( ), strset( ), and strnset( ). */
#include <stdio.h>
#include <string.h>
char str[] = "This is C programming class.";
void main( )
{
    clrscr( );
    printf("\nThe original string: %s", str);
    printf("\nCalling strrev( ): %s", strrev(str));
    printf("\nCalling strrev( ) again: %s", strrev(str));
    printf("\nCalling strnset( ): %s", strnset(str, '!', 5));
    printf("\nCalling strset( ): %s", strset(str, 'c'));
    getch( );
}
OUTPUT:
The original string: This is C programming class.
Calling strrev( ): .ssalc gnimmargorp C si sihT
Calling strrev( ) again: This is C programming class.
Calling strnset( ): !!!!!is C programming class.
Calling strset( ): cccccccccccccccccccccccccccc
```

CHARACTER OPERATIONS

Character Analysis

The header file **CTYPE.H** contains the prototypes for a number of functions that test characters, returning TRUE or FALSE depending on whether the character meets a certain condition. For example, is it a letter or is it a numeral? The **is*xxxx*( )** functions are actually macros, defined in **CTYPE.H.**

The **is*xxxx*( )** macros all have the same prototype: **int is*xxxx*(int ch);**

In the preceding line, ch is the character being tested. The return value is TRUE (nonzero) if the condition is met or FALSE (zero) if it isn't. The following table lists the complete set of **is*xxxx*( )** macros.

FUNCTION	**TESTS FOR**
islower( )	Lowercase letter
isupper( )	Uppercase letter
isascii()	Standard ASCII character (between 0 and 127).
isalpha( )	Uppercase or lowercase letter
isalnum( )	Uppercase or lowercase letter or a digit
iscntrl( )	Control character
isprint( )	Any printing character including space
isgraph( )	Any printing character except space
isdigit( )	Decimal digit ('0' to '9')
isxdigit( )	Hexadecimal digit ('0' to '9', 'A' to 'F', 'a' to 'f')
isblank( )	Standard blank characters (space, '\t')
isspace( )	Whitespace character (space, '\n', '\t', '\v', '\r', '\f')
ispunct( )	Printing character for which isspace( ) and isalnum( ) return false

Converting Characters

The standard library also includes two conversion functions that you get access to through **<ctype.h>.** The **toupper( )** function converts from lowercase to uppercase, and the **tolower( )** function does the reverse. Both functions return either the converted character or the same character for characters that are already in the correct case. You can therefore convert a string to uppercase using this statement: **for(int i = 0 ; (buffer[i] = toupper(buffer[i])) != '\0' ; i++);**

This loop will convert the entire string to uppercase by stepping through the string one character at a time, converting lowercase to uppercase and leaving uppercase characters unchanged. The loop stops when it reaches the string termination character '\0'.

/* program to count number of words, number of characters and number of special characters in a given string*/

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
#include<ctype.h>
void main( )
{
    char string[100];
    int nch=0,nw=0,nsch=0,i=0;
    clrscr( );
    printf("Enter any string:");
    gets(string);
    while(string[i]!='\0')
    {
        if(tolower(string[i])>='a'&& tolower(string[i])<='z')
            nch++;
        else if(string[i]==' ')
        {
            nw++;
            nch++;
        }
        else
            nsch++;
        i++;
    }
    printf("\nNumber of characters in the given string are: %d",nch);
    printf("\nNumber of special characters in the given string are: %d",nsch);
    printf("\nNumber of words in the given string are: %d",nw+1);
    getch( );
}
```

OUTPUT:

```
Enter any string: this is an example program to count number of characters
Number of characters in the given string are: 56
Number of special characters in the given string are: 0
Number of words in the given string are: 10
```

/* program to count number of vowels, number of special characters and number of consonants in a given string*/

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
#include<ctype.h>
void main( )
{
    char string[100],ch;
    int nchr=0,nv=0,nc=0,nsch=0,i=0;
    clrscr( );
    printf("Enter any string:");
    gets(string);
    while(string[i]!='\0')
    {
       ch=string[i];
       if(isalpha(ch))
       {
         nchr++;
         if(tolower(ch)=='a'||tolower(ch)=='e'||tolower(ch)=='i'||tolower(ch)=='0'||tolower(ch)=='u')
           nv++;
         else
           nc++;
       }
       else
       {
         nsch++;
         nchr++;
        }
        i++;
    }
    printf("\nNumber of special characters(including space) in the given string are: %d",nsch);
    printf("\nNumber of vowels in the given string are: %d",nv);
    printf("\nNumber of consonants in the given string are: %d",nc);
    printf("\nTotal Number of characters in the given string are: %d",nchr);
    getch( );
}
```

```
OUTPUT:
Enter any string:c programming & datastructures b-tech first year

Number of special characters(including space) in the given string are: 8
Number of vowels in the given string are: 11
Number of consonants in the given string are: 29
Total Number of characters in the given string are: 48
```

Write a program to sort the set of strings in an alphabetical order.

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
#include<stdbool.h>
#include<string.h>
#define BUFFER_LEN 100 /* Length of input buffer */
#define NUM_P 100 /* maximum number of strings */
void main( )
{
    char buffer[BUFFER_LEN]; /* space to store an input string */
    char *pS[NUM_P] = { NULL }; /* Array of string pointers */
    char *pTemp = NULL; /* Temporary pointer */
    int i = 0; /* Loop counter */
    bool sorted = false; /* Indicated when strings are sorted */
    int last_string = 0; /* Index of last string entered */
    printf("\nEnter successive lines, pressing Enter at the"
    " end of each line.\nJust press Enter to end.\n\n");
    while((*fgets(buffer, BUFFER_LEN, stdin) != '\n') && (i < NUM_P))
    {
       pS[i] = (char*)malloc(strlen(buffer) + 1);
       if(pS[i]==NULL) /* Check for no memory allocated */
       {
          printf(" Memory allocation failed. Program terminated.\n");
          return 1;
       }
       strcpy(pS[i++], buffer);
    }
    last_string = i; /* Save last string index */
    /* Sort the strings in ascending order */
    while(!sorted)
    {
        sorted = true;
        for(i = 0 ; i<last_string-1 ; i++)
            if(strcmp(pS[i], pS[i + 1]) > 0)
```

```
                {
                 sorted = false; /* We were out of order */
                 pTemp= pS[i]; /* Swap pointers pS[i] */
                 pS[i] = pS[i + 1]; /* and */
                 pS[i + 1] = pTemp; /* pS[i + 1] */
                }
        }
        /* Displayed the sorted strings */
        printf("\nYour input sorted in order is:\n\n");
        for(i = 0 ; i<last_string ; i++)
        {
            printf("%s\n", pS[i] );
            free( pS[i] );
            pS[i] = NULL;
        }
        getch( );
}
```

OUTPUT:

```
Enter successive lines, pressing Enter at the end of each line.
Just press Enter to end.
this is a c program to sort strings
write this carefully
c programming
arrays
pointers

Your input sorted in order is:
arrays
c programming
pointers
this is a c program to sort strings
write this carefully
```

/* Write a C program to count the lines, words and characters in a given text*/

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
void main( )
{
        char line[81], ctr;
        int i,c,end = 0,characters = 0,words = 0,lines = 0;
        clrscr( );
        printf("ENTER  TEXT.\n");
        printf("GIVE ONE SPACE AFTER EACH WORD.\n");
        printf("WHEN COMPLETED, PRESS ENTER.\n\n");
        while( end==0)
        {
                /* Reading a line of text */
                c = 0;
                while((ctr=getchar())!='\n')
                        line[c++] = ctr;
                line[c] = '\0';
                /* counting the words in a line */
                if(line[0]=='\0')
                        break ;
                else
                {
                        words++;
                        for(i=0; line[i] != '\0';i++)
                                        if(line[i] ==' '||line[i]=='\t')
                                                words++;
                }
                /* counting lines and characters */
                lines = lines +1;
                characters = characters + strlen(line);
        }
        printf("Number of lines = %d\n", lines);
        printf("Number of words = %d\n", words);
        printf("Number of characters = %d\n", characters);
        getch( );
}
```

```
OUTPUT:
ENTER  TEXT.
GIVE ONE SPACE AFTER EACH WORD.
WHEN COMPLETED, PRESS ENTER.
algorithm is a step by step procedure
flow chart is diagramatic representation
c is a programming language
we know c programming very well
we can write c programs for any problems

Number of lines = 5
Number of words = 31
Number of characters = 174
```

Write a C program to find whether a given string is palindrome or not

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
void main( )
{
      char str[20];
      int len,flag=1,i,l;
      clrscr( );
      printf("Enter a string : ");
      scanf("%s",str);
      len = strlen(str);
      l=len-1;
      for(i=0;i<len/2;i++)
      {
         if(str[i]!=str[l-i])
         flag = 0;
      }
      if(flag==1)
         printf("\n Given string is palindrome");
      else
         printf("\n Given string is not a palindrome");
    getch( );
}
```

```
OUTPUT 1:
Enter a string : program
 Given string is not a palindrome

OUTPUT 2:
Enter a string : madam
 Given string is palindrome

OUTPUT 3:
Enter a string : liril
 Given string is palindrome
```

Program to read five cities and sort them and print sorted list of cities in alphabetical order

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
void main( )
{
      char city[5][20],temp[20];
      int i,n=0;
      clrscr( );
      printf("enter the names of five cities...\n\n");
      for(i=0;i<5;i++)
      scanf("%s",&city[i]);
      printf("sorted list of cities...\n\n");
     while(!n)
     {
       n=1;
      for(i=0;i<4;i++)
      {
         if(strcmp(city[i],city[i+1])>0)
          {
             n=0;
             strcpy(temp,city[i]);
             strcpy(city[i],city[i+1]);
             strcpy(city[i+1],temp);
          }
      }
      }
      for(i=0;i<5;i++)
      printf("\n%s",city[i]);
      getch( );
}
```

OUTPUT:

```
enter the names of five cities...

nandyal
kurnool
hyderabad
vijayawada
guntur

sorted list of cities...

guntur
hyderabad
kurnool
nandyal
vijayawada
```

Eg: Write a C program to read string from keyboard and display it using character pointer

```
 #include<stdio.h>
#include<conio.h>
void main( )
{
    char name[20],*ch;
    clrscr( );
    printf("Enter your name: ");
    gets(name);
    ch=name; /* stores the base address of name*/
    while(*ch!='\0')
    {
      printf("%c",*ch);
      ch++;
    }
    getch( );
}
```

```
OUTPUT:
Enter your name: nageswara reddy
nageswara reddy
```

Eg: Write a C program to find length of a given string including and excluding spaces using pointer

```
#include<stdio.h>
#include<conio.h>
void main( )
{
    char str[50],*s;
    int p=0,q=0;
    clrscr( );
    printf("Enter a string: ");
    gets(str);
    s=str;
    while(*s!='\0')
    {
      printf("%c",*s);
      p++;
      s++;
      if(*s==32)/* ASCII value of space is 32*/
      q++;
    }
    printf("\nLength of string including spaces : %d", p);
    printf("\nLength of string without spaces : %d", p-q);
    getch( );
}
```

```
OUTPUT:
Enter a string: this is a program to find length of a string
this is a program to find length of a string
Length of string including spaces : 44
Length of string without spaces : 35
```

UNIT-IV

Pointers: Fundamentals, Pointer Declarations, Passing pointer to a function, Pointers and one dimensional array, Dynamic memory allocation, Operations on pointers, Pointers and multi dimensional arrays, Arrays of pointers, Passing functions to other functions, More about pointer declarations.

Structures and Unions: Defining a structure, Processing a structure, User defined data type (typedef), Structures and Pointers, Passing structures to functions, Unions.

File Handling: Why files, Opening and closing a data file, Reading and Writing a data file, Processing a data file, Unformatted data files, Conceptof binary files, Accessing the file randomly (using fseek).

Additional Features: Register variables, Bitwise operations, Bit Fields, Enumerations, Command line parameters, More about Library functions, Macros, The C Preprocessor

INTRODUCTION:

OBJECTIVE: One of the powerful features of C is its ability to access the memory variables by their memory addresses. A pointer data type is mainly used to hold memory address. Pointers are useful to work with memory addresses, to pass values as arguments to functions, to allocate memory dynamically and to effectively represent complex data structures. Since arrays store data sequentially in memory, pointers allow a convenient and powerful manipulation of array elements.

A pointer is a derived data type in C. It is built from one of the fundamental data types available in C. **Pointers contain memory address as their values.** Pointers are one of the most distinct and exciting features of C language. It has added power and flexibility to the language.

A pointer is a variable which contains the address in memory of another variable. We can have a pointer to any variable type. The ***unary*** *or* ***monadic*** *operator* **&** *gives the "****address of a variable****"****.*** *The* ***indirection*** *or dereference operator (*) gives the "contents of an object* ***pointed to*** *by a pointer".*

Need of pointers:

- Basically arrays are static. It means that the maximum possible size of the array has to be declared before its use (i.e., at compile time).It is not always possible to guess the maximum size of an array, because for some applications we need the size of an array to be changed during the program execution. This can be achieved by using the pointers. Pointers allows memory allocation and de-allocation dynamically.
- Pointers are used for establishing links between data elements or objects for some complex data structures such as stacks, queues, linked lists, binary trees and graphs.

FEATURES OF POINTERS

Pointers offer a number of benefits to the programmers:

- Pointers are more efficient in handling arrays and data tables.
- Pointers can be used to written multiple values from a function via function arguments.
- Pointers permit reference to functions and there by facilitating passing of functions as arguments to other functions.
- The use of pointer arrays to character string results in saving of data storage space in memory.
- Pointers allow C to support dynamic memory management.
- Pointers provide an efficient tool for manipulating dynamic data structures such as structures, linked lists, queues, stacks and trees.
- Pointers reduce length and complexity of programs. They increase the execution speed and reduce the program execution time because data is manipulated with the address i.e., direct access to memory location.

POINTER DECLARATION:

Pointer variable can be declared as:

data_type *PointerName;

Here the ***** tells that variable **PointerName** is pointer type variable i.e. it holds the address of another variable specified by the **data-type**. PointerName needs a memory location .PointerName points to a variable of type data_type.

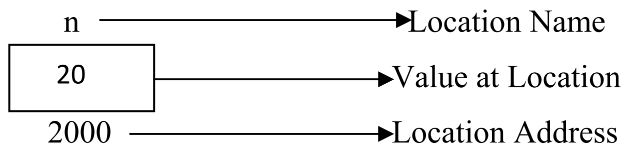
Consider the following declaration.

int n =20;

This declaration tells the C compiler to:

1. Reserve space in memory to hold the integer value.
2. Associate the name with this memory location.
3. Store the value 20 at this location.

We may represent **n's** location in the memory by the following memory map:



int *p; /* pointer declaration*/

p=&n; /*pointer initialization*/

1. In the above statement **p** is an integer pointer and it tells to the compiler that it holds the address of any integer variable. Second statement says that **p** is storing address of variable **n.**
2. The indirection operator (*) is also called dereference operator. When a pointer is dereferenced, the value at that address stored by the pointer is retrieved.
3. Normal variables provide direct access to their own values whereas a pointer provides indirect access to the values of the variable whose address it stores.
4. The indirection operator (*) is used in two distinct ways with pointers, declaration and dereference.
5. When a pointer is declared, the star indicates that it is a pointer, not a normal variable.
6. The **&** is the address operator and it represents the address of the variable. The %u is used with printf( ) function for printing the address of the variable

/*Program to print address and the value of a variable by using '&' and '*' operators */

```
#include<stdio.h>
#include<conio.h>
void main( )
{
    int n;
    clrscr( );
    printf("Enter a number:\n");
    scanf("%d",&n);
    printf ("address of n is: %u \n ", &n);
    printf ("value of n is: %d \n", n);
    printf ("value of n is: %d",*(&n));
    getch( );
}
```

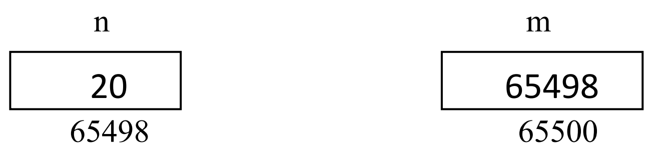
```
OUTPUT:
Enter a number:
20
address of n is: 2293572        /* value varies*/
value of n is: 20
value of n is: 20
```

In the first printf ( ) statement **'&'** is used it is C's **address of operator**. The expression &n returns the address of the variable n, which in this it is 2293572. The third printf ( ) statement we used other pointer operator '*' called **'value at address'** operator. It returns the value stored at a particular address. The 'value at address' operator is also called as '**indirection'** operator. The above program says the value of *(&n) is same as n.

In the above example &n returns the address of n, if we desire this address can be collected in a variable by saying

m=&n;

But remember that m is not an ordinary variable like any other integer variable. It is a variable which contains the address of another variable (n in this case). The following memory map would illustrate the contents of n and m.



As you can see **n's** value is 20 and m's value is **n's** address. Here we can't use m in a program with out declaring it. And since m is a variable which contains the address of n, it is declared as **int * m;**

This declaration tells compiler that **m** will be used to store the address of an integer value. In other words **m** points to an integer.

/* Program to print address and the value of a variable by using & and * operators */

```
#include<stdio.h>
#include<conio.h>
void main( )
{
     int n, *m;
     m=&n;
    clrscr( );
     printf("Enter a number:\n");
     scanf("%d",&n);
     printf ("address of n is: %u \n", &n);
     printf ("address of n  is:%u\n", m);
     printf ("address of m is: %u\n", &m);
     printf ("value of m is: %u\n", m);
     printf ("value of n is: %d\n ",n);
     printf ("value of n is: %d\n",*(&n));
     printf ("value of n is: %d",*m);
     getch( );
}
```

OUTPUT:

```
Enter a number:
20
address of n is: 2293572
address of n is:2293572
address of m is: 2293568
value of m is: 2293572
value of n is: 20
 value of n is: 20
value of n is: 20
```

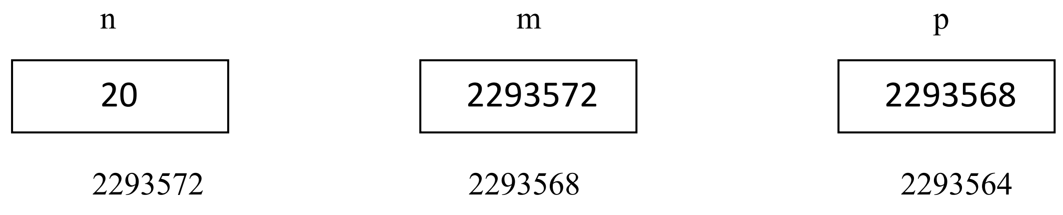
The concept of pointer can be further extended. Pointer we know is a variable which contains address of another variable. Now this variable itself could be another pointer. Thus we have a pointer which contains another pointer's address.

/* Program to print address and the value of a variable by using &, *and **operators */

```
#include<stdio.h>
#include<conio.h>
main( )
{
    int n, *m,**p;
    m=&n;
    p=&m;
    clrscr( );
    printf("Enter a number:\n");
    scanf("%d",&n);
    printf ("address of n is: %u \n ", &n);
    printf ("address of n is: %u \n", m);
    printf ("address of n is: %u \n", *p);
    printf ( "address of m is :%u \n", &m);
    printf ("address of m is: %u \n", p);
    printf ("address of p is: %u \n" ,&p);
    printf ("value of m is: %u \n", m);
    printf ("value of p is: %u \n", p);
    printf ("value of n is: %d \n", n);
    printf ("value of n is: %d \n",*(&n));
    printf ("value of n is %d\n ", *m);
    printf ("value of n is: %d \n", **p);
    getch( );
}
```

```
OUTPUT:
Enter a number:
20
address of n is: 2293572
address of n is: 2293572
address of n is: 2293572
address of m is :2293568
address of m is: 2293568
address of p is: 2293564
value of m is: 2293572
value of p is: 2293568
value of n is: 20
value of n is: 20
value of n is 20
 value of n is: 20
```

The following memory map would help you in tracing out how the program prints the above output



PASSING POINTERS TO A FUNCTION

- Pointers are often passed to a function as arguments. This allows data items within the calling portion of the program to be accessed by the function, altered within the function, and then returned to the calling portion of the program in altered form. We refer to this use of pointers as passing arguments by ***reference*** (or by ***address*** or by ***location)***
- When an argument is passed by reference, however (i.e., when a pointer is passed to a function), the ***address*** of a data item is passed to the function. The contents of that address can be accessed freely, either within the function or within the calling routine.
- Thus, the use of a pointer as a function argument permits the corresponding data item to be altered globally from within the function.
- When pointers are used as arguments to a function, some care is required with the formal argument declarations within the function. Specifically, formal pointer arguments that must each be preceded by an asterisk. Function prototypes are written in the same manner. If a function declaration does not include variable names, the data type of each pointer argument must be followed by an asterisk.
- It is possible to pass aportion of an array, rather than an entire array, to a function. To do so, the address of the first array element to be passed must be specified as an argument. The remainder of the array, starting with the specified array element, will then be passed to the function.
- A function can also return a pointer to the calling portion of the program. To do *so,* the function definition and any corresponding function declarations must indicate that the function will return a pointer. This is accomplished by preceding the function name by an asterisk. The asterisk must appear in both the function definition and the function declarations

POINTERS AND ARRAYS:

- In C, there is a strong relationship between pointers and arrays, strong enough that pointers and arrays should be discussed simultaneously. Any operation that can be achieved by array subscripting can also be done with pointers. The pointer version will in general be faster but, at least to the uninitiated, somewhat harder to understand.
- The declaration **int a[10];** defines an array of size 10, that is, a block of 10 consecutive objects named a[0], a[1], ...,a[9].

  The notation **a[i]** refers to the i-th elementof the array.

  If **pa** is a pointer to an integer, declared as **int *pa;**

  then the assignment **pa = &a[0];**

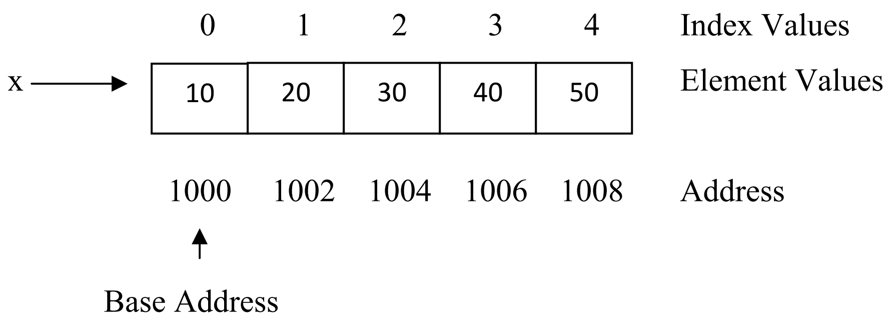
  sets pa to point to element zero of a; that is, pa contains the address of a[0].

  Now the assignment **x = *pa;** will copy the contents of a[0] into x.
- If pa points to a particular elementof an array, then by definition pa+1 points to the next element, pa+i points i elements after pa, and pa-i points i elements before.

  Thus, if pa points to a[0], *(pa+1) refers to the contents of a[1], pa+i is the address of a[i], and *(pa+i) is the contents of a[i].
- These remarks are true regardless of the type or size of the variables in the array a. The meaning of “adding 1 to a pointer”, and by extension, all pointer arithmetic, is that pa+1 points to the nextobject, and pa+i points to the i-th object beyond pa.
- The correspondence between indexing and pointer arithmetic is very close. By definition, the value of a variable or expression of type array is the address of element zero of the array. Thus after the assignment **pa = &a[0]; pa** and **a** have identical values. Since the name of an array is a synonym for the location of the initial element, the assignment pa=&a[0] can also be written as **pa = a;**

  A reference to a[i] can also be written as *(a+i). In evaluating a[i], C converts it to *(a+i) immediately; the two forms are equivalent.
- Applying the operator & to both parts of this equivalence, it follows that &a[i] and a+i are also identical: a+i is the address of the i-th element beyond a. If pa is a pointer, expressions might use it with a subscript; pa[i] is identical to *(pa+i). In short, an array-and-index expression is equivalent to one written as a pointer and offset.
- There is one difference between an array name and a pointer that must be kept in mind. A pointer is a variable, so **pa=a and pa++ are legal**. But an array name is not a variable; constructions like **a=pa and a++ are illegal.**

Example: int x[5] = {10,20,30,40,50};



From this, x = &x[0] = &x all are referred to the address location 1000.

Array subscripting is defined in terms of pointer arithmetic operations.

Let

```
int x[3], i=0;
&x[0] = x+0            x[0] = *(x+0)
&x[1] = x+1            x[1] = *(x+1)
&x[2] = x+2            x[2] = *(x+2)
i.e., &x[i] = x+i      x[i] = *(x+i)
```

Program to display array elements with their addresses using array name as a pointer.

```
#include<stdio.h>
#include<conio.h>
void main( )
{
    int num[44]={10,25,35,45}, i;
    clrscr( );
    for(i=0;i<4;i++)
    {
      printf("num[%d]=%d",i,*(num+i));
      printf("\t%8u\n",num+i);
    }
    getch( );
}
```

OUTPUT:

```
num[0]=10      2293392
num[1]=25      2293394
num[2]=35      2293396
num[3]=45      2293398
```

In this program the statement ***(num+i)** is same as **num[i];**

What is the difference between an *array* and a *pointer*?

An array is pretty much the same as a pointer. When we pass an *array* to a function we are really passing a *pointer* to the startof the array. This is why we can change the value of array elements within a function – because really, we were passing a pointer all along.

There are a few differences between pointers and arrays:

1) Arrays have memory initialised for them – and therefore we can start using them right away. Pointers must be initialised to be used.

```
int a[12];
int *b;
a[3]= 5;   /*sets 4th elementof a to 5 */
*a= 3;     /*sets 1st elementof a to 3 – same as a[0]= 3; */
b[3]= 5;   /* Error - b is not initialised */
*b= 3;     /* Error – b is not initialised */
```

2) We can set a pointer to point to something else – we cannot do this with an array. An array must always point to the block of memory it was initialised with.

```
int a[12];
int *b;
b= a;      /* Fine, sets b to point to a – note that because a pointer  is basically an array we don't
need b= &a; */
a= b;     /* Error – we can't make a point at something else */
```

3) We have multidimensional arrays. There is no such thing as a multi-dimensional pointer.

```
int a[12][12];
int *b;
a[0][0]= 3;  /* this is fine*/
b[0][0]= 3;  /* this is always an error */
```

ARITHMETIC OPERATIONS WITH POINTERS

C pointer is an address which is a numeric value. Therefore, you can perform arithmetic operations on a pointer just as you can a numeric value.

Incrementing Pointers: When you *increment* a pointer, you are increasing its value. For example, when you increment a pointer by 1, pointer arithmetic automatically increases the pointer's value so that it points to the next array element. In other words, C knows the data type that the pointer points to (from the pointer declaration) and increases the address stored in the pointer by the size of the data type.

Suppose that ptrtoint is a pointer variable to some elementof an int array. If you execute the statement **ptrtoint++;** the value of ptrtoint is increased by the size of type int (usually 2 bytes), and ptrtoint now points to the next array element. Likewise, if ptrtofloat points to an elementof a type float array, the statement **ptrtofloat++;** increases the value of ptrtofloat by the size of type float (usually 4 bytes).

The same holds true for increments greater than 1.If you add the value *n* to a pointer, C increments the pointer by *n* array elements of the associated data type. Therefore, **ptrtoint += 4;** increases the value stored in ptrtoint by 8 (assuming that an integer is 2 bytes), so it points four array elements ahead. Likewise, **ptrtofloat += 10;** increases the value stored in ptrtofloat by 40 (assuming that a float is 4 bytes), so it points 10 array elements ahead.

Decrementing Pointers

The same concepts that apply to incrementing pointers hold true for decrementing pointers. *Decrementing* a pointer is actually a special case of incrementing by adding a negative value. If you decrement a pointer with the -- or -= operators, pointer arithmetic automatically adjusts for the size of the array elements.

EX: **Using pointer arithmetic and pointer notation to access array elements**

```
/* Demonstrates using pointer arithmetic to access array elements with pointer notation. */
#include <stdio.h>
#include<conio.h>
#define MAX 10
void main( )
{
        int i_array[MAX] = { 0,1,2,3,4,5,6,7,8,9 };
        int *i_ptr, count;
        float f_array[MAX] = { .0, .1, .2, .3, .4, .5, .6, .7, .8, .9 };
        float *f_ptr;
        i_ptr =i_array;
        f_ptr =f_array;
        clrscr( );
        for (count = 0; count < MAX; count++)
                printf("%d\t%f\n", *i_ptr++, *f_ptr++);
        getch( );
}
```

OUTPUT:

```
0       0.000000
1       0.100000
2       0.200000
3       0.300000
4       0.400000
5       0.500000
6       0.600000
7       0.700000
8       0.800000
9       0.900000
```

<u>**Other Pointer Manipulations**</u>

The only other pointer arithmetic operation is called ***differencing,*** which refers to subtracting two pointers. If you have two pointers to different elements of the same array, you can subtract them and find out how far apart they are. Again, pointer arithmetic automatically scales the answer so that it refers to array elements. Thus, if ptr1 and ptr2 point to elements of an array (of any type), the following expression tells you how far apart the elements are: **ptr1 - ptr2**

Pointer comparisons are valid only between pointers that point to the same array. Under these circumstances, the relational operators ==, !=, >, <, >=, and <= work properly. Lower array elements (that is, those having a lower subscript) always have a lower address than higher array elements. Thus, if ptr1 and ptr2 point to elements of the same array, the comparison **ptr1 < ptr2** is true if ptr1 points to an earlier member of the array than ptr2 does.

Many arithmetic operations that can be performed with regular variables, such as multiplication and division, don't make sense with pointers. The C compiler doesn't allow them. For example, if ptr is a pointer, the statement **ptr *= 2;** generates an error message.

POINTER OPERATIONS:

Operation	**Description**
Assignment	You can assign a value to a pointer. The value should be an address, obtained with the address-of operator (&) or from a pointer constant (array name).
Indirection	The indirection operator (*) gives the value stored in the pointed-to location.
Address of	You can use the address-of operator to find the address of a pointer, so you can have pointers to pointers
Incrementing	You can add an integer to a pointer in order to point to a different memory location.
Decrementing	You can subtract an integer from a pointer in order to point to a different memory location.
Differencing	You can subtract two pointers that point to elements of same array and find out how far they are.
Comparison	Valid only with two pointers that point to the same array.

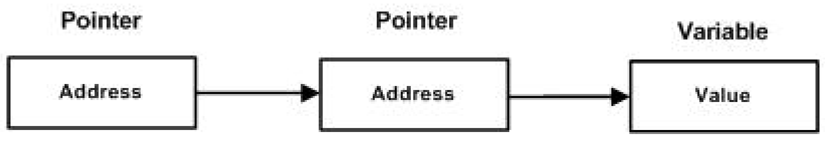
These permissible operations are summarized below.

1. A pointer variable can be assigned the address of an ordinary variable (e.g., pv = &v).
2. A pointer variable can be assigned the value of another pointer variable (e.g., pv = px) provided both pointers point to objects of the same data type .
3. A pointer variable can be assigned a null (zero) value (e.g., pv = NULL, where NULL is a symbolic constant that represents the value 0).
4. An integer quantity can be added to or subtracted fkom a pointer variable (e.g., pv + 3, ++pv)
5. One pointer variable can be subtracted from another provided both pointers point to elements of the same array.
6. Two pointer variables can be compared provided both pointers point to objects of the same data type.

Other arithmetic operations on pointers are not allowed. Thus, a pointer variable cannot be multiplied by a constant; two pointer variables cannot be added; and so on. Also, you are again reminded that an ordinary variable cannot be assigned an arbitrary address (i.e., an expression such as &x cannot appear on the left side of an assignment statement).

POINTER TO POINTERS

A pointer to a pointer is a form of multiple indirection, or a chain of pointers. Normally, a pointer contains the address of a variable. When we define a pointer to a pointer, the first pointer contains the address of the second pointer, which points to the location that contains the actual value as shown below.



A variable that is a pointer to a pointer must be declared as such. This is done by placing an additional asterisk (*) in frontof its name. For example, following is the declaration to declare a pointer to a pointer of type int:

int **var;

When a target value is indirectly pointed to by a pointer to a pointer, accessing that value requires that the asterisk operator be applied twice, as is shown below in the example:

```
#include<stdio.h>
#include<conio.h>
void main ( )
{
    int  var;
    int  *ptr;
    int  **pptr;
    clrscr( );
    var = 3000;
    ptr = &var;/* take the address of var */
    pptr = &ptr;/* take the address of ptr using address of operator & */
    printf("Value of var = %d\n", var );
    printf("Value of var through *ptr = %d\n", *ptr );
    printf("Value of var through **pptr = %d\n", **pptr);
    printf("address of var is %u\n ",&var);
    printf("value at ptr is %d;address of ptr is %u\n",ptr,&ptr);
    printf("value at pptr is %d;address of pptr is %u\n",pptr,&pptr);
    getch( );
}
```

```
OUTPUT:
Value of var = 3000
Value of var through *ptr = 3000
Value of var through **pptr = 3000
address of var is 2293572
 value at ptr is 2293572;address of ptr is 2293568
value at pptr is 2293568;address of pptr is 2293564
```

VOID POINTERS (GENERIC POINTERS)

When a variable is declared as being a pointer to type void it is known as a generic pointer. Since you cannot have a variable of type void, the pointer will not point to any data and therefore cannot be dereferenced. It is still a pointer though, to use it you just have to cast it to another kind of pointer first. Hence the term Generic pointer. This is very useful when you want a pointer to point to data of different types at different times.

Try the following code to understand Generic Pointers.

```
#include <stdio.h>
#include<conio.h>
void main( )
{
    int i;
    char c;
    void *ptr;
    i =6;
    c ='a';
    ptr=&i;
    printf("ptr points to the integer value %d:\n",*(int*)ptr);
    ptr =&c;
    printf("ptr now points to the character %c:\n",*(char*)ptr);
    getch( );
}
```

```
OUTPUT:
ptr points to the integer value: 6
ptr now points to the character: a
```

NOTE-1: Here in first print statement, ptr is prefixed by ***(int*)**. This is called type casting in C language. Type is used to cast a variable from one data type to another data type to make it compatible to the lvalue.

NOTE-2: lvalue is something which is used to left side of a statement and in which we can assign some value. A constant can't be an lvalue because we cannot assign any value in contact. For example x = y, here x is lvalue and y is rvalue.

NULL POINTER

Uninitialized pointer variable is initialized to a value in such a way that it is certain not to point to any objector function. A pointer initialized in this manner is called a NULL pointer.

A null pointer is a special type of pointer that cannot points to anywhere. Null pointer is assigned by using the predefined constant NULL; which is defined by several header files including stdio.h, stdlib.h and alloc.h.

Example: **int *p=NULL;**

```
/* EXAMPLE PROGRAM FOR NULL POINTER */

#include<stdio.h>
main( )
{
	int *p;
	clrscr();
	p=NULL;
	printf("\nValue :%d",*p);
}
```

DYNAMIC MEMORY ALLOCATION FUNCTIONS

The memory allocation may be classified as static memory allocation and dynamic memory allocation.

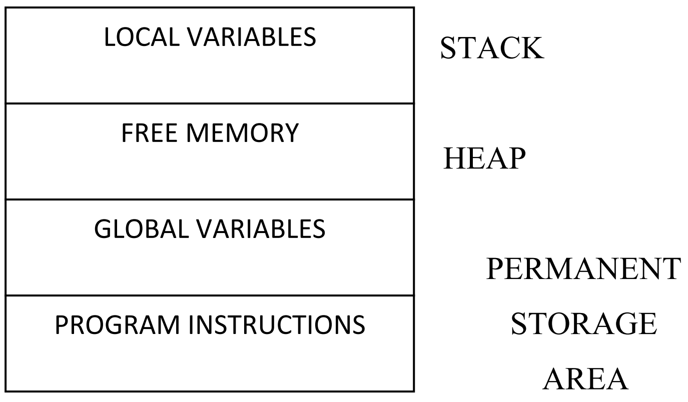
Static memory allocation: Memory for the variables is created at the time of compilation is known as static memory.

Dynamic memory allocation: Memory for the variables is allocated at the time of execution of the program is called dynamic memory. The following functions are used for dynamic memory allocation which are defined in **stdlib.h** and **alloc.h** header files.

1. malloc() 2. calloc() 3. realloc() 4. free()

malloc(), calloc() and realloc() are memory allocation functions and free() is a memory releasing function.

Memory allocation process



Pointers are an extremely flexible and powerful tool for programming over a wide range of applications. The majority of programs in C use pointers to some extent. C also has a further facility that enhances the power of pointers and provides a strong incentive to use them in the code; it permits memory to be allocated dynamically when the program executes. Allocating memory dynamically is possible only because of pointers available.

When you explicitly allocate memory at runtime in a program, space is reserved for you in a memory area called the **heap**. There's another memory area called the **stack** in which space to store function arguments and local variables in a function is allocated. When the execution of a function is finished, the space allocated to store arguments and local variables is freed. The memory in the heap is controlled by the programmer. When you allocate memory on the heap, it is up to you to keep track of when the memory you have allocated is no longer required and free the space you have allocated to allow it to be reused.

<u>Memory allocation with malloc ( )</u> void *malloc(size_t *size*);

The simplest standard library function that allocates memory at runtime is called malloc( ). You need to include the <stdlib.h> header file in your program when you use this function. When you use the malloc( ) function, you specify the number of bytes of memory that you want allocated as the argument. The function returns the address of the first byte of memory allocated in response to your request. A typical example of dynamic memory allocation might be this: **int *pNumber = (int *)malloc(100);**

In the above statement we are requesting 100 bytes of memory and assigned the address of this memory block to pNumber. pNumber, will point to the first int location at the beginning of the 100 bytes that were allocated.

If there is insufficient memory in the heap to satisfy the request, **malloc( )** returns a null pointer. It is always important to verify that the return value is not null before attempting to use it. Attempting to use a null pointer will usually result in a system crash.

Using the sizeof Operator in Memory Allocation: The previous example is all very well, but you don't usually deal in bytes; you deal in data of type int, type double, and so on. It would be very useful to allocate memory for 75 items of type int, for example.You can do this with the following statement:

pNumber = (int *) malloc(75*sizeof(int));

sizeof is an operator that returns an unsigned integer that is countof the number of bytes required to store its argument. It will accept a type keyword such as intor float as an argument between parentheses, in which case the value it returns will be the number of bytes required to store an item of that type. It will also accept a variable or array name as an argument. With an array name as an argument, it returns the number of bytes required to store the whole array.

```
/* EXAMPLE PROGRAM FOR MALLOC ( ) FUCNTION */
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
void main( )
{
        int *p,i,n;
        clrscr( );
        printf("\nEnter how many numbers:");
        scanf("%d",&n);
        p=(int*)malloc(n*sizeof(int));
        printf("\nEnter %d Elements:",n);
        for(i=0;i<n;i++)
```

```
                scanf("%d",p+i);
        printf("\nArray Elements Are:");
        for(i=0;i<n;i++)
                printf("%3d",*(p+i));
   getch( );
}
```

OUTPUT:

```
Enter how many numbers:5
Enter 5 Elements: 1 2 3 4 5
Array Elements Are:  1  2  3  4  5
```

Memory Allocation with the calloc( ) Function **void *calloc(size_t *num*, size_t *size*);**

The calloc( ) function that is declared in the <stdlib.h> header offers a couple of advantages over the malloc( ) function. First, it allocates memory as an array of elements of a given size, and second, it initializes the memory that is allocated so that all bits are zero. The calloc( ) function requires you to supply two argument values, the number of elements in the array, and the size of the array element, both arguments being of type **size_t**. The function still doesn't know the type of the elements in the array so the address of the area that is allocated is returned as type void *.

Here's how you could use calloc( ) to allocate memory for an array of 75 elements of type int:

```
        int *pNumber = (int *) calloc(75, sizeof(int));
```

The return value will be NULL if it was not possible to allocate the memory requested, so you should still check for this. This is very similar to using malloc( ) but the big plus is that you know the memory area will be initialized to zero.

```
#include<conio.h>
#include<stdlib.h>
void main( )
{
        int *p,i,n;

        printf("\nEnter how many numbers:");
        scanf("%d",&n);
        p=(int*)calloc(n,sizeof(int));
        printf("\nEnter %d Elements:",n);
        for(i=0;i<n;i++)
                scanf("%d",p+i);
```

```
            printf("\nArray Elements Are:");
            for(i=0;i<n;i++)
               printf("%3d",*(p+i));
   getch( );
}
```

OUTPUT:

```
Enter how many numbers:5
Enter 5 Elements: 1 2 3 4 5
Array Elements Are:  1  2  3  4  5
```

<u>Reallocating memory</u> **void *realloc(void ****ptr*, **size_t** ***size*);**

The realloc( ) function enables you to reuse memory that you previously allocated using malloc( ) or calloc( ) (or realloc( )).The realloc( ) function expects two argument values to be supplied: a pointer containing an address that was previously returned by a call to malloc( ), calloc( ) or realloc( ), and the size in bytes of the new memory that you want allocated.

The realloc( ) function releases the previously allocated memory referenced by the pointer that you supply as the first argument, then reallocates the same memory area to fulfill the new requirement specified by the second argument. Obviously the value of the second argument should not exceed the number of bytes that was previously allocated. If it is, you will only get a memory area allocated that is equal to the size of the previous memory area.

<u>Note:</u>

I. The new memory block may or may not be begin at the same place as the old one. In case, it is not able to find additional space in the same region, it will create the same in an entirely new region and moves the contents of the old block into the new block.

II. If the function is unsuccessful to allocate the memory space, it returns a NULL pointer and the original block is lost.

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
void main( )
{
            int *p,i,n;
            clrscr( );
            printf("\nEnter how many numbers:");
```

```
          scanf("%d",&n);
          p=(int*)malloc(n*sizeof(int));
          printf("\nEnter %d Elements:",n);
          for(i=0;i<n;i++)
                    scanf("%d",p+i);
          p=(int*)realloc(p,(n+2)*sizeof(int));
          printf("\nEnter %d Elements:",n+2);
          for(i=0;i<n+2;i++)
                    scanf("%d",p+i);
          printf("\nArray Elements Are:");
          for(i=0;i<n+2;i++)
                    printf("%3d",*(p+i));
   getch( );
}
```

OUTPUT:

```
Enter how many numbers:3
Enter 3 Elements:1 2 3
Enter 5 Elements:4 5 6 7 8
Array Elements Are:  4  5  6  7  8
```

<u>Releasing Dynamically Allocated Memory</u> **void free (void ****ptr***);**

When you allocate memory dynamically, you should always release the memory when it is no longer required. Memory that you allocate on the heap will be automatically released when your program ends, but it is better to explicitly release the memory when you are done with it, even if it's just before you exit from the program. In more complicated situations, you can easily have a **memory leak**. A memory leak occurs when you allocate some memory dynamically and you do not retain the reference to it, so you are unable to release the memory. This often occurs within a loop, and because you do not release the memory when it is no longer required, your program consumes more and more of the available memory and eventually may occupy it all.

Of course, to free memory that you have allocated using malloc( ) or calloc( ), you must still be able to use the address that references the block of memory that the function returned. To release the memory for a block of dynamically allocated memory whose address you have stored in the pointer pNumber, you just write the statement**: free (pNumber);**

The free ( ) function has a formal parameter of type void *, and because any pointer type can be automatically converted to this type, you can pass a pointer of any type as the argument to the function. As long as pNumber contains the address that was returned by malloc( ) or calloc( ) when the memory was allocated, the entire block of memory that was allocated will be freed for further use.

If you pass a null pointer to the free( ) function the function does nothing. You should avoid attempting to free the same memory area twice, as the behavior of the free( ) function is undefined in this instance and therefore unpredictable. You are most at risk of trying to free the same memory twice when you have more than one pointer variable that references the memory you have allocated, so take particular care when you are doing this.

Here are some basic guidelines for working with memory that you allocate dynamically:

• Avoid allocating lots of small amounts of memory. Allocating memory on the heap carries some overhead with it, so allocating many small blocks of memory will carry much more overhead than allocating fewer larger blocks.

• Only hang on to the memory as long as you need it. As soon as you are finished with a block of memory on the heap, release the memory.

• Always ensure that you provide for releasing memory that you have allocated. Decide where in you code you will release the memory when you write the code that allocates it.

• Make sure you do not inadvertently overwrite the address of memory you have allocated on the heap before you have released it; otherwise your program will have a memory leak. You need to be especially careful when allocating memory within a loop.

POINTERS AND TWO DIMENSIONAL ARRAYS:

A matrix can represent two dimensional elements of an array. In one dimensional array the expression *(a+i) or *(pa+i) represents the [i] element. Similarly, an element in a two dimensional array can be represented by the pointer expression as: ***(*(a+i) +j) or *(*(p+i) +j)**

A two dimensional array **a[i][j]** has i rows and j columns; The base address of array a is &a[0][0] and starting at this address the compiler allocates contiguous space for all the elements row-wise. That is, the first elementof second row is placed immediately after the last elementof first row and so on.

Pointers in two dimensional arrays;

pa	→ pointer to first row
pa+i	→ pointer to i th row
*(pa+i)	→ pointer to first element in the i th row
*(pa+i)+j	→ pointer to j th element in the i th row
((pa+i)+j)	→ value stored in the cell (i,j) (i th row j th column

Program to display two dimensional array elements .Use array name itself as pointer.

```
#include<stdio.h>
#include<conio.h>
void main( )
{
    int a[3][3]={{1,2,3},{4,5,6},{7,8,9}},i,j;
    clrscr( );
    printf("VALUES OF EACH ARRAY ELEMENTS\n");
    for(i=0;i<3;i++)
    {
      for(j=0;j<3;j++)
                printf("a[%d][%d]=%d\t",i,j,*(*(a+i)+j));
     printf("\n");
    }
    printf("ADDRESS OF EACH ARRAY ELEMENT\n");
    for(i=0;i<3;i++)
    {
        for(j=0;j<3;j++)
                printf("a[%d][%d]=%u\t",i,j,*(a+i)+j);
      printf("\n");
     }
    getch( );
}
```

```
OUTPUT:
VALUES OF EACH ARRAY ELEMENTS
a[0][0]=1      a[0][1]=2      a[0][2]=3
a[1][0]=4      a[1][1]=5      a[1][2]=6
a[2][0]=7      a[2][1]=8      a[2][2]=9
ADDRESS OF EACH ARRAY ELEMENT
a[0][0]=2293520 a[0][1]=2293522 a[0][2]=2293524
a[1][0]=2293526 a[1][1]=2293528 a[1][2]=2293530
a[2][0]=2293532 a[2][1]=2293534 a[2][2]=2293536
```

ARRAY OF POINTERS:

Since pointers are variables themselves, they can be stored in arrays just as other variables can. This array is nothing but collection of addresses. Here, we store addresses of variables for which we have to declare an array as a pointer.

Program to store addresses of different elements of an array – using array of pointers.

```
#include<stdio.h>
#include<conio.h>
void main( )
{
    int arr[5]={1,2,3,4,5},i;
    int *arrp[5];
    clrscr( );
    for(i=0;i<5;i++)
    {
      arrp[i]=&arr[i];
    }
    printf("ELEMENT\tVALUE\tADDRESS\n");
    for(i=0;i<5;i++)
    {
     printf("arr[%d]\t%d\t%u\n",i,*arrp[i],arrp[i]);
    }
    getch( );
}
```

OUTPUT:

```
ELEMENT   VALUE   ADDRESS
arr[0]      1      2293536
arr[1]      2      2293538
arr[2]      3      2293540
arr[3]      4      2293542
arr[4]      5      2293544
```

EG:

```
#include<stdio.h>
#include<conio.h>
void main( )
{
    int arr1[5]={1,2,3,4,5},arr2[5]={6,7,8,9,10};
    int arr3[5]={11,12,13,14,15},arr4[5]={16,17,18,19,20},i,j;
    int *arrp[4];
    clrscr( );
    arrp[0]=&arr1[0];
    arrp[1]=&arr2[0];
    arrp[2]=&arr3[0];
    arrp[3]=&arr4[0];
    printf("VALUE and ADDRESS\n");
    for(i=0;i<4;i++)
    {
        for(j=0;j<5;j++)
                printf("%d-%u\t",*(*(arrp+i)+j),*(arrp+i)+j);
        printf("\n");
    }
    getch( );
}
```

POINTER TO FUNCTION

One of the powerful features of C is the *function pointer*. In C, a function itself is not a variable, but it is possible to define pointers to functions, which can be assigned, placed in arrays, passed to functions, returned by functions, and so on.

A function has a physical location in memory that can be assigned to a pointer. This address is the entry pointof the function and it is the address used when the function is called. Once a pointer points to a function, the function can be called through that pointer. Function pointers also allow functions to be passed as arguments to other functions.

The address of a function by using the function's name without any parentheses or arguments. (This is similar to the way an array's address is obtained when only the array name, without indexes, is used.)

EG: **int (*p) (const char *, const char *);**

This declaration tells the compiler that **p** is a pointer to a function that has two **const char *** parameters, and returns an **int** result. The parentheses around p are necessary in order for the compiler to properly interpret this declaration.

Program to call a function using pointer

```
#include<stdio.h>
#include<conio.h>
void display( );
void main( )
{
    void  (*ptr)( );          /* declaration of pointer to a function*/
    ptr=display;
    clrscr( );
    (*ptr)( );                /* function call using pointer*/
    printf("address of function display( ) is %u\n",ptr);
    getch( );
}
void display( )
{
   printf("function call using pointer to function\n");
}
```

OUTPUT:

```
function call using pointer to function
address of function display( ) is 4199136
```

MORE ABOUT POINTER DECLARATIONS

Pointer declarations can become complicated, and some care is required in their interpretation. This is especially true of declarations that involve functions or arrays.

One difficulty is the dual use of parentheses. In particular, parentheses are used to indicate functions, and they are used for nesting purposes (to establish precedence) within more complicated declarations. Thus, the declaration

int *p(int a) ; indicates a function that accepts an integer argument, and returns a pointer to an integer. On the other hand, the declaration

int (*p) (int a) ; indicates a ***pointer*** to a ***function*** that accepts an integer argument and returns an integer. In this last declaration, the first pair of parentheses is used for nesting, and the second pair is used to indicate a function.

The interpretation of more complex declarations can be increasingly troublesome. For example, consider the declaration

int *(*p)(int (* a) [ ]) ;

In this declaration, (***p)** (. . .) indicates a pointer to a function. Hence, **int** * (***p)** (. . .) indicates a pointer to a function that returns a pointer to an integer. Within the last pair of parentheses (the function's argument specification), **(*a) [ ]** indicates a pointer to an array. Therefore, **int** (***a)** [ ] represents a pointer to an array of integers. Putting the pieces together, (***p)** (**int** (***a)** [ ]) represents a pointer to a function whose argument is a pointer to an array of integers. And finally, the entire declaration **int *(*p)(int (* a) [ ]) ;** represents a pointer to a function that accepts a pointer to an array of integers as an argument, and returns a
pointer to an integer.

Remember that a left parenthesis immediately following an identifier name indicates that the identifier represents a function. Similarly, a left square bracket immediately following an identifier name indicates that the identifier represents an array. Parentheses that identify functions and square brackets that identify arrays have a higher precedence than the unary indirection operator. Therefore, additional parentheses are required when declaring a pointer to a function or a pointer to an array.

The following example provides a number of illustrations.

Several declarations involving pointers are shown below. The individual declarations range from simple to complex.

- int *p;   /* p is a pointer to an integer quantity */
- int *p[10]; /* p is a 10-element array of pointers to integer quantities */
- int (*p) [10];   /* p is a pointer to a 10-element integer array */
- int *p (void) ;   / * p is a function that returns a pointer to an integer quantity */
- int p(char *a); /* p is a function that accepts an argument which is a pointer to a character and returns an integer quantity */

- int *p(char a*); /* p is a function that accepts an argument which is a pointer to a character and returns a pointer to an integer quantity */
- int (*p)(char *a); /* p is a pointer to a function that accepts an argument which is a pointer to a character returns an integer quantity */
- int (*p(char * a)) [10] ; /* p is a function that accepts an argument which is a pointer to a character returns a pointer to a 10-element integer array */
- int p(char (* a) [ ]) ; /* p is a function that accepts an argument which is a pointer to a character array returns an integer quantity */
- int p(char * a [ ]) ;   /* p is a function that accepts an argument which is an array of pointers to characters returns an integer quantity */
- int *p(char a [ ]) ; /* p is a function t h a t accepts an argument which is a character array returns a pointer to an integer quantity */
- int *p(char (* a) [ ]) ; /* p is a function that accepts an argument which is a pointer to a character array returns a pointer to an integer quantity */
- int *p(char * a [ ]) ; /* p is a function that accepts an argument which is an array of pointers to characters returns a pointer to an integer quantity */
- int (*p)(char (* a) [ ]) ; /* p is a pointer to a function that accepts an argument which is a pointer to a character array returns an integer quantity */
- int *(*p)(char (* a) [ ]) ; / * p is pointer to a function that accepts an argument which is a pointer to a character array returns a pointer to an integer quantity */
- int *(*p)(char * a [ ]) ; / * p is a pointer to a function that accepts an argument which is an array of pointers to characters returns a pointer to an integer quantity */
- int (* p [10]) (v o i d) ; / * p is a 10-element array of pointers to functions; each function returns an integer quantity */
- int (*p[10])(char a); /* p is a 10-element array of pointers to functions; each function accepts an argument which is a character, and returns an integer quantity */
- int * (* p [10]) (c h a r a); /* p is a 10-element array of pointers to functions; each function accepts an argument which is a character, and returns a pointer to an integer quantity */
- int * (* p [10]) (c h a r *a); /* p is a 10-element array of pointers to functions; each function accepts an argument which is a pointer to a character, and returns a pointer to an integer quantity */

Differences between arrays and pointers:

ARRAY	**POINTER**
1. For an array, compiler allocates memory space automatically and used by the array.	1. For a pointer, it is explicitly assigned to point to an allocated space.
2. It can not be resized.	2. It can be resized using realloc() function.
3. It cannot be reassigned.	3. It can be reassigned.
4. sizeof(arrayname) gives the number of bytes of the memory occupied by the array.	4. sizeof(pointervaraible) returns only 2 bytes.

STRUCTURES AND UNIONS:

Structures provide a way to organize related data. Unlike arrays, structures allow organization of collection of variables. With different data types structures are very useful in creating data structures. Unions also provide a way to organize related data, but only one item within the union can be used at any time.

INTRODUCTION:

We have seen arrays can be used to represent a group of data items that belong to the same type, such as int or float. However we cannot use an array if we want to represent a collection of data items of different types using the single name. C supports the constructed data type known as structures, a mechanism for packing data of different types**.**

"A structure is a collection of one or more variables, possibly of different types, grouped together under a single name for convenient handling. (Structures are called "records" in some languages, notably Pascal.) ". Structures help to organize complicated data, particularly in large programs, because they permit a group of related variables to be treated as a unit instead of as separate entities.

EX:

Time	Seconds (int), Minutes (int), Hours (float)
Date	Day (int), Month (string), Year (int)
Book	Author (string), Title (string), Price (float)
Address	Name (string), Doornumber (string), Street (string), City (string)

Structures help to organize complex data in a more meaningful way. It is a powerful concept that we may often need to use in our program design.

FEATURES OF STRUCTURES:

To copy elements of the one array to another array of same data type elements are copied one by one. It is not possible to copy all the elements at a time. Where as in structure it is possible to copy the contents of all structure elements of different data types to another structure variable of its type using assignment(=) operator. It is possible because the structure elements are stored in successive memory locations.

Nesting of structures is possible i.e. one can create structure with in the structure. Using these feature one can handle complex data types.

It is also possible to pass structure elements to a function. This is similar to passing an ordinary variable to a function. One can pass individual structure elements or entire structure by value or address.

It is also possible to create structure pointers. For creating a pointer pointing to structure elements it requires → operator.

STRUCTURE DEFINITION:

A structure in C is heterogeneous compound data type, similar to the records of database and PASCAL. It is collection of logically related data items grouped together under a single name called structure tag. The data items that make up a structure are known as its members, components, or fields and can be of different type.

STRUCTURE DECLARATION:

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

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

In this declaration, **struct** is a required keyword; **tag** is a name that identifies structures of this type.

The individual **members** can be ordinary variables, pointers, arrays or other structures. The member names within a particular structure must be distinct from one another, though a member name can be same as the name of a variable defined outside of the structure.

A storage class, however, cannot be assigned to an individual member, and individual members cannot be initialized within a structure-type declaration. For example:

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

we can now declare the structure variable s1 and s2 as follows:

struct student s1, s2;

s1 and s2 are structure type variables whose composition is identified by the tag student.

It is possible to combine the declaration of the structure composition with that of the structure variable as shown below.

The tag is optional in this situation.

```
struct student {
          char name [80];
          int roll_no;
          float marks;
}s1, s2;
```

The s1, s2, are structure variables of type student. Since the variable declarations are now combined with the declaration of the structure type, the tag need not be included. As a result, the above declaration can also be written as

```
        struct{
                        char name [80];
                        int roll_no;
                        float marks ;
                } s1, s2;
```

Eg2: Ex:

```
            struct book_bank
                {
                        char title[25];
                        char author[20];
                        int pages;
                        float price;
                };
```

we can now declare the structure variable **struct book_bank book1,book2,book3;**

```
        struct book_bank
        {
                char  title[25];
                char author[20];
                int  pages;
                float price;
        } book1, book2, book3;
```

The use of tag name is optional.

For example,

```
        struct
        {       char title[25];
                char author[20];
                int  pages;
                float price;
        }book1, book2, book3;
```

declares book1,book2,and book3 as structure variables representing three books, but does not include a tag name for later use in declarations.

STRUCTURE INITIALIZATION:

Like other data types a structure variable can be initialized. However a structure must be declared as static.

```
main( )
{
        static struct
        {
                int age;
                float height;
        } student= {20,180.75};
………………
}
```

This assigns the value 20 to student.age and 180.75 to student.height.

Suppose you want to initialize more than one structure variable:

```
main( )
{
        struct  st_record
        {
                int  age;
                float  height;
        };
static  struct st_record student1={20,180.75};
static  struct st_record student2={22,177.25};
…………………..
}
```

Another method is to initialize a structure variable outside the function

```
struct st_record
{         int   age;
          float  height;
}student1={20,180.75};
main( )
{
   static struct st_record student2={22,177.25};
   ………..
 }
```

ACCESSING STRUCTURE ELEMENTS:

The members of a structure are usually processed individually, as separate entities. Therefore, we must be able to access the individual structure members. The link between a member and a variable are established using a member **operator "."** which is also known as **dot operator** or period **operator**. A structure member can be accessed by writing variable.member **name**.

This period (.) is an operator; it is a member of the highest precedence group, and its associativity is left-to-right.

If we want to print the detail of a member of a structure then we can write as printf("%s",book1.title); or printf("%d", book1.pages) and so on. More complex expressions involving the repeated use of the period operator may also be written.

We assign values to the members of structure variables as follows

```
strcpy ( book1.title, "Let us C");
book1.pages=250;
book1.price=255.00
```

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

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

EG: Write a program to define a structure and initialize its member variables

```
#include<stdio.h>
#include<conio.h>
void main( )
{
    struct book
    {   char bookname[30];
        int pages;
        float price;
    };
    struct book bk1={"c programming",300,250};
    clrscr( );
    printf("\n BOOK NAME : %s",bk1.bookname);
    printf("\n NO OF PAGES : %d",bk1.pages);
    printf("\n BOOK PRICE: %6.2f",bk1.price);
    getch( );
}
```

OUTPUT:

```
BOOK NAME : c programming
NO OF PAGES : 300
BOOK PRICE: 250.00
```

EG: WRITE A PROGRAM TO COPY STRUCTURE ELEMENTS FROM ONE OBJECT TO ANOTHER OBJECT

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
void main( )
{
    struct disk
    {
      char co[15];
      float type;
      int price;
    };
    struct disk d1={"SONY",1.44,29};
    struct disk d2,d3;
    strcpy(d2.co,d1.co);
    d2.type=d1.type;
    d2.price=d1.price;
    d3=d2;
    clrscr( );
    printf("\nvalues using d1:  %s\t%f\t%d",d1.co,d1.type,d1.price);
    printf("\nvalues using d2:  %s\t%f\t%d",d2.co,d2.type,d2.price);
    printf("\nvalues using d3:  %s\t%f\t%d",d3.co,d3.type,d3.price);
    getch( );
}
```

OUTPUT:

```
values using d1:  SONY  1.440000       29
values using d2:  SONY  1.440000       29
values using d3:  SONY  1.440000       29
```

EG: WRITE A PROGRAM TO READ VALUES USING SCANF ( ) AND ASSIGN THEM TO STRUCTURE VARIABLE.

```
#include<stdio.h>
#include<conio.h>
void main( )
{
    struct book
    {
        char bookname[30];
        int pages;
        float price;
    };
    struct book bk1;
    clrscr( );
    printf("Enter book name: ");
    scanf("%s",bk1.bookname);
    printf("Enter no of pages : ");
    scanf("%d",&bk1.pages);
    printf("Enter price: ");
    scanf("%f",&bk1.price);
    printf("\n BOOK NAME : %s",bk1.bookname);
    printf("\n NO OF PAGES : %d",bk1.pages);
    printf("\n BOOK PRICE: %6.2f",bk1.price);
    getch( );
}
```

```
OUTPUT:
Enter book name: cpds
Enter no of pages : 300
Enter price: 250

 BOOK NAME : cpds
 NO OF PAGES : 300
 BOOK PRICE: 250.00
```

Ex: ***/* DEFINE A STRUCTURE TYPE, STRUCT PERSONAL, THAT WOULD CONTAIN PERSON NAME, DATE OF JOINING AND SALARY, USING THIS STRUCTURE, WRITE A PROGRAM TO READ THIS INFORMATION FOR ONE PERSON FROM THE KEYBOARD AND PRINT THE SAME ON THE SCREEN */***

```
#include<stdio.h>
#include<conio.h>
struct  personal
{
char  name[20];
int  day;
char  month[10];
int  year;
int salary;
};
void main( )
{
struct  personal  person;
clrscr( );
printf("ENTER
PERSONALDETAILS:\n\n");
printf(" Enter person name : ");
scanf("%s", person.name);
printf("Enter a person joining  day : ");
scanf("%d", &person.day);
printf("Enter a person joining  month(in words): ");
scanf("%s", person.month);
printf("Enter a person joining  year: ");
scanf("%d", &person.year);
printf("Enter a person salary: ");
scanf("%d", &person.salary);
printf("\n\n person's name is : %s\n", person.name);
printf("\n\n person's  joining day  is : %d\n", person.day);
printf("\n\n person's joining month is : %s\n", person.month);
printf("\n\n person's joining year is : %d\n", person.year);
printf("\n\n person's salary is : %d\n", person.salary);
getch( );
}
```

```
OUTPUT:
ENTER PERSONAL DETAILS:
 Enter person name : KNREDDY
Enter a person joining  day : 2
Enter a person joining  month(in words):
JANUARY
Enter a person joining  year: 2010
Enter a person salary: 25000

 person's name is : KNREDDY

 person's  joining day  is : 2

 person's joining month is : JANUARY

 person's joining year is : 2010

 person's salary is : 25000
```

COMPARISION OF STRUCTURE VARIABLE:

Two variables of the same structure type can be compared the same way as ordinary variables.

If person1 and person2 belongs to the same structure, then the following operations are valid:

person1 = person2 ------ assigns person1 to person2

However the statements such as

person1= =person2

person1! =person2

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

/*PROGRAM TO ILLUSTRATE THE COMPARISON OF STRUCTURE VARIABLES */

```
#include<stdio.h>
#include<conio.h>
void main( )
{
    int x;
    struct  stuclass
    {      int no;
          char name[20];
          float marks;
    };
    stuclass stu1={111,"Naga",72.50};
    stuclass stu2={222,"Eswar",67.00};
    stuclass stu3;
    stu3=stu2;
    clrscr( );
    if(stu2.no==stu3.no && stu2.marks==stu3.marks)
      printf("\n student2 and student3 are same\n");
    else
      printf("\n student2 and student3 are different\n");
    getch( );
}
OUTPUT:  student2 and student3 are same
```

ARRAYS OF STRUCTURES:

We may declare an array of structures, each element of the array representing a structure variable. For example

struct class student[100];

defines an array called **'student'** that consists of 100 elements. Each element is defined to be of the type **struct class**. Consider the following declaration

```
struct marks
{
        int eng;
        int cds;
        int phy;
};
main( )
{
        struct marks student[3]={45,76,87},{78,68,79},{34,23,14};
        …………….
        …………….
}
```

This declares the **student** as an array of three elements **student [0], student [1]** and **student [2].** Each member of student array is a structure variable with three members.

An array of structures is stored inside the memory in the same way as a multidimensional array. The array student looks as shown in fig:

student[0]	subject 1	45
	subject 2	76
	subject 3	87
student[2]	subject 1	78
	subject 2	68
	subject 3	79
student[3]	subject 1	34
	subject 2	23
	subject 3	14

Ex: /*WRITE A PROGRAM TO CALCULATE THE SUBJECT-WISE AND STUDENT-WISE TOTALS AND STORE AS A PART OF THE STRUCTURE*/

```
#include<stdio.h>
#include<conio.h>
struct marks
{
        int eng;
        int cds;
        int phy;
        int tot;
};
main( )
{
int i;
static struct marks
student[3]={{45,67,81,0},{75,53,69,0},{57,36,71,0}};
static struct marks total;
for(i=0;i<3;i++)
{
student[i].tot=student[i].eng+student[i].cds+student[i].phy;
total.eng=total.eng+student[i].eng;
total.cds=total.cds+student[i].cds;
total.phy=total.phy+student[i].phy;
total.tot=total.tot+student[i].tot;
}
printf(" STUDENT TOTAL  \n\n");
for(i=0;i<3;i++)
printf(" stu[%d]   : %d\n",i+1,student[i].tot);
printf("\n\nSUBJECT TOTAL\n\n");
printf("English :%d\ncds :%d\nphy %d\n", total.eng,
                    total.cds, total.phy ) ;
printf("\nGrand total :  %d\n",total.tot);
getch( );
}
```

OUTPUT:

```
STUDENT TOTAL

 stu[1]   : 193
 stu[2]   : 197
 stu[3]   : 164


SUBJECT TOTAL

English :177
cds :156
phy :221

Grand total :  554
```

ARRAYS WITHIN STRUCTURES:

C permits the use of array as structure members. We can use single or multi-dimensional array of type int or float.

```
Ex:         struct marks
{
        int no;
        int sub[5];
        float fee;
}stu[10];
```

Here, the member **'subs'** containing 5 elements. These elements can be accessed using appropriate subscripts. For example, stu[1].sub[2]; would refer to the marks of third subject by the second student.

Ex: Rewrite the program of above example using an array member to represent the three subjects.

```
/***********************************************************************
                    Arrays within a structure
***********************************************************************/
```

```
#include<stdio.h>
#include<conio.h>
struct marks
        {
                int sub[3];
                int tot;
        };
void main( )
{
        static struct marks student[3]={45,67,81,0,75,53,69,0,57,36,71,0};
        static struct marks total;
        int i,j;
        for(i=0;i<=2;i++)
        {
                for(j=0;j<=2;j++)
                {
                        student[i].tot+=student[i].sub[j];
                        total.sub[j]+=student[i].sub[j];
```

```
                    }
                    total.tot+=student[i].tot;
          }
          printf("\nSTUDENT TOTAL\n\n");
          for(i=0;i<=2;i++)
                    printf("student[%d]=%d\n",i+1,student[i].tot);
          printf("\nSUBJECT TOTAL\n\n");
          for(j=0;j<=2;j++)
                    printf("subject[%d]=%d\n",j+1,total.sub[j]);
          printf("\nGrand Total  =%d\n", total.tot);
          getch( );
}
```

OUTPUT:

```
STUDENT TOTAL

student[1]=193
student[2]=197
student[3]=164

SUBJECT TOTAL

subject[1]=177
subject[2]=156
subject[3]=221

Grand Total  =554
```

STRUCTURES WITHIN STRUCTURES:

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

```
struct salary
{       char name[20];
        char dept[20];
        int basic_pay;
        int dearness_allowance;
        int house_rent_allowance;
        int city_allowance;
} employee;
```

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

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

The ‘salary’ structure contains a member named ‘allowance’, which itself is a structure with three members. The members contained in the inner structure namely, **dearness, house_rent,** and **city** can be referred to as

employee.allowance.dearness;

employee.allowance.house_rent;

employee.allowance.city;

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

The following being invalid:

employee.allowance	------------	actual member is missing
employee.house_rent	------------	inner structure variable is missing

Ex: **Program to enter full name and date of birth of a person and display the same. Use nested structures**

```
#include<stdio.h>
#include<conio.h>
void main( )
{
    struct name{
        char first[20];
        char second[20];
        char last[20];
        };
    struct birthdate{
        int day;
        int month;
        int year;
        };
    struct data{
        struct name nm;
        struct birthdate bd;
        };
    struct data r;
    clrscr( );
    printf("Enter first   second   last name \n");
    scanf("%s%s%s",r.nm.first,r.nm.second,r.nm.last);
    printf("Enter bitrh date(day month year)");
    scanf("%d%d%d",&r.bd.day,&r.bd.month,&r.bd.year);
    printf("NAME : %s  %s  %s \n",r.nm.first,r.nm.second,r.nm.last);
    printf("BIRTH DATE :%d-%d-%d",r.bd.day,r.bd.month,r.bd.year);
    getch( );
}
```

OUTPUT:

```
Enter first   second   last name
ram sham pande
Enter bitrh date(day month year)12 12 2012
NAME : ram  sham  pande
BIRTH DATE :12-12-2012
```

POINTER TO STRUCTURE

The way we can have a pointer pointing to an int, or a pointer pointing to a char, similarly we can have a pointer pointing to the struct. Such pointers are known as ***'structure pointers'***.

```
EG:     struct book {
                        char title[25];
                        char author[25];
                        int pages;
                        };
            struct book *ptr;
```

In the above example *ptr is pointer to structure book. The syntax for using pointer with member is as given below.

1) ptr→name   2) ptr→author   3) ptr→pages

By executing these three statements starting address of each member can be estimated.

/*WRITE A PROGRAM TO DECLARE POINTER TO STRUCTURE AND DISPLAY THE COMTENTS OF THE STRUCTURE */

```
#include<stdio.h>
#include<conio.h>
void main( )
{
   struct book
         {
          char  title[25];
          char author[25];
          int pages;
          };
    struct book b={"C Notes","knreddy",250};
    struct book *ptr;
    ptr=&b;
    printf("\n%s\t%s\t%d\n",b.title,b.author,b.pages);

printf("\n%s\t%s\t%d\n",ptr→title,ptr→author,ptr→pages);
    getch( );
}
```

OUTPUT:

```
C Notes knreddy 250

C Notes knreddy 250
```

The first printf( ) is as usual. The second printf( ) however is peculiar. We cannot use ptr.tittle, ptr.author and ptr.no because ptr is not a structure variable but a pointer to a structure, and the dot operator requires a structure variable on its left. In such cases C provides an operator → called an *arrow operator* to refer the structure elements.

/*WRITE A PROGRAM TO DECLARE POINTER AS MEMBER OF STRUCTURE AND DISPLAY THE CONTENTS OF THE STRUCTURE */

```
#include<stdio.h>
#include<conio.h>
void main( )
{
   struct boy
         {
          char  *name;
          int *age;
          float *height;
          };
   struct boy *ptr;
   char nm[10]="mahesh";
   int ag=20;
   float ht=5.40;
   ptr->name=nm;
   ptr->age=&ag;
   ptr->height=&ht;
   clrscr( ):
   printf("\nNAME: %s\nAGE: %d\nHEIGHT: %f\n",ptr->name,*ptr->age,*ptr->height);
   getch( );
}
```

OUTPUT:

```
NAME: mahesh
AGE: 20
HEIGHT: 5.400000
```

STRUCTURE AND FUNCTIONS:

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

- The **first method** is to pass each member of the structure as an actual argument of the function call. The actual arguments are then read independently like ordinary variables. This is the most elementary approach and becomes unmanageable and inefficient when the structure size is large.
- The **second method** involves passing of a copy of the entire structure to the called function. Since the function is working on a copy of the structure, any changes to the structure members within the function are not reflected in the original structure. It is, therefore, necessary for the function to return the entire structure back to the calling function.
- The **third approach** employs a concept called ***pointers*** to pass the structure as an argument. In this case, the address location of the structure is passed to the called function. The function can access indirectly the entire structure and work on it. This is similar to the way, arrays are passed to functions. This method is more efficient as compared to the second one.

/* WRITE A PROGRAM TO PASS STRUCTURE ELEMENTS TO FUNCTION */

```
#include<stdio.h>
#include<conio.h>
void display(char *t,char *n, int p);
void main( )
{
    struct book{
        char title[10];
        char author[10];
        int pages;
        };
    struct book b1={"c notes","knreddy",250};
    display(b1.title,b1.author,b1.pages);
    getch( );
}
void display(char *t,char *n, int p)
{
     printf("\ntitle : %s\nauthor: %s\npages: %d\n",t,n,p);
}
```

```
OUTPUT:
title : c notes
author: knreddy
pages: 250
```

/* PROGRAM TO PASS ADDRESS OF A STRUCTURE VARIABLE TO FUNCTION */

```
#include<stdio.h>
#include<conio.h>
struct book{
            char  title[25];
            char author[25];
            int pages;
            };
void display(struct book *b2);
void main( )
{
   struct book b1={"C Notes","knreddy",250};
   clrscr( );
   display(&b1);
   getch( );
}
void display(struct book *b2)
{
  printf("TITLE:%s\nAUTHOR: %s\nPAGES: %d\n",b2->title,b2->author,b2->pages);
}
```

```
OUTPUT:
TITLE: C Notes
AUTHOR: knreddy
PAGES: 250
```

/* PROGRAM TO PASS ENTIRE STRUCTURE TO USER DEFINED FUNCTION*/

```
#include<stdio.h>
#include<conio.h>
struct book{
            char  title[25];
            char author[25];
            int pages;
            };
void display(struct book b2);
void main( )
{
  struct book b1={"C Notes","knreddy",250};
   display(b1);
   getch( );
}
void display(struct book b2)
{
  printf("TITLE:%s\nAUTHOR: %s\nPAGES: %d\n",b2.title,b2.author,b2.pages);
}
```

```
OUTPUT:
TITLE:C Notes
AUTHOR: knreddy
PAGES: 250
```

Ex: Write a simple program to illustrate the method of sending an entire structure as a parameter to a function.

```
#include<stdio.h>
#include<conio.h>
 struct stores
 {
   char name[20];
   float price;
   int qty;
 };
 struct stores update(struct stores,float,int); /*function prototype*/
 void main( )
 {
   float mul(struct stores);
   float p_inc,val;
   int q_inc;
   struct stores item={"pen",5.50,10};
   clrscr( );
   printf("\nInitial datails of item:\n");
   printf("Name      :%s\n",item.name);
   printf("Price     :%f\n",item.price);
   printf("Quantity  :%d\n",item.qty);
   printf("\nInput increment values :");
   printf("\nprice increment and quantity increment\n");
   scanf("%f%d",&p_inc,&q_inc);
   item=update(item,p_inc,q_inc);     /*function call*/
   printf("\nUpdate values of item:\n");
   printf("Name      :%s\n",item.name);
   printf("Price     :%f\n",item.price);
   printf("Quantity  :%d\n",item.qty);
   val=mul(item);                         /*function call*/
   printf("\nValue of the item=%f\n",val);
   getch( );
}
```

```
struct stores update(struct stores prod,float p,int q)
{
  prod.price+=p;
  prod.qty+=q;
  return(prod);
}
float mul(struct stores stock)
{
   return(stock.price * stock.qty);
}
OUTPUT:
Initial datails of item:
Name      :pen
Price     :5.500000
Quantity  :10
Input increment values :
price increment and quantity increment
2
3
Update values of item:
Name      :pen
Price     :7.500000
Quantity  :13
Value of the item=97.500000
```

Important points:

- The called function must be declared for its type, appropriate to the data type it is expected to return.
- The structure variable used as the actual argument and the corresponding formal argument in the called function must be of the same **struct** type.
- The return statement is necessary only when the function is returning some data. The *expression* may be any simple variable or structure variable or an expression using simple variables.
- When a function returns a structure, it must be assigned to a structure of identical type in the calling function.
- The called function must be declared in the calling function, if it is placed after the calling function.

UNIONS

Unions are a concept borrowed from structures and therefore follow the same syntax as structures. However there is major distinction between them in terms of storage. In structures each member has its own storage location, whereas all the members of a union use the same location. It can handle only one member at a time.

General format:

```
union tag{     type var1;
               type var2;
               .
          };
```

Ex:

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

This declares a variable code of type union item. The union contains three members each with a different data type. However we can use only one of them at a time. This is due to the fact that only one location is allocated for a union variable, irrespective of its size.

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

In the above declaration the member x requires 4 bytes which is the largest among the members.

ACCESSING UNION ELEMENTS:

To access a union member we can use the same syntax that we used in the structure members.

Ex: code.m; code.x;

Pointers to remember:

- **struct** is a key word which shows the start of a structure.
- A structure can be passed to as a single variable to function.
- A union can also be stored as a member of the structure.
- Pointers can also store the address of heterogeneous types of elements i.e., structures.
- **DON'T** try to initialize more than the first union member.
- **DO** remember which union member is being used. If you fill in a member of one type and then try to use a different type, you can get unpredictable results.
- The size of a union is equal to its largest member.

```
/* Program to find size of union */
#include<stdio.h>
#include<conio.h>
#include<string.h>
union Data
{
        int i;
        float f;
        char  str[20];
};
main( )
{
        union Data info;
        printf( "Memory size occupied by info : %d\n", sizeof(info));
        getch( );
}
OUTPUT:
Memory size occupied by info : 20
```

UNION OF STRUCTURES

In a union, each piece of data starts at the same memory location and occupies at least a part of the same memory. Thus, when a union is shared by two or more different data types, only one piece of data can be in memory.

Consider, for example, an address book that contains both the company name and individual names. When we store a company name, the name has only one field. On the other hand, when we store an individual's name, the name has at least three parts-first name, middle name, last name. If we want to have only one name field in our address book, we need to use a union to store the name.

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
typedef struct{
        char first[20];
        char init;
        char last[20];            }person;
```

```
typedef struct{
        char type;
        union{
            char company[20];
            person p;
            }un;
              }name;
void main( )
{
      int i;
      name business={'c',"abc company"};
      name pname;
      name names[2];
      pname.type='p';
      strcpy(pname.un.p.first,"nageswar");
      strcpy(pname.un.p.last,"reddy");
      pname.un.p.init='k';
      names[0]=business;
      names[1]=pname;
      for(i=0;i<2;i++)
              switch(names[i].type)
              {   case  'c': printf("company: %s\n",names[i].un.company);
                                break;
                  case  'p': printf("pname:%c%s%s\n",names[i].un.p.init, names[i].un.p.first,
                                                                               names[i].un.p.last );
                             break;
                  default : printf("Error in type ");
               }
      getch( );
}
OUTPUT:
company: abc company
pname: k nageswar reddy
```

```
/*Write a program to use structure within union. Display the contents of structure elements. */
#include<stdio.h>
#include<conio.h>
void main( )
        {
                struct student  {
                        char name[30];
                        char sex;
                        int rollno;
                        float percentage;
                };
                union details {
                        struct student st;
                };
                union details set;
                printf("Enter details:\n");
                printf("Enter name: ");
                scanf("%s", set.st.name);
                printf("Enter rollno: ");
                scanf("%d", &set.st.rollno);
                printf("Enter sex: ");
                scanf(" %c",&set.st.sex);
                printf("Enter percentage: ");
                scanf("%f",&set.st.percentage);
                printf("The student details are:\n");
                printf("Name : %s", set.st.name);
                printf("\nRollno : %d", set.st.rollno);
                printf("\nSex : %c", set.st.sex);
                printf("\nPercentage        :        %f",
set.st.percentage);
getch( );
}
```

```
OUTPUT:
Enter details:
Enter name: knr
Enter rollno: 2315
Enter sex: m
Enter percentage: 80
The student details are:
Name : knr
Rollno : 2315
Sex : m
Percentage : 80.000000
```

```
/* EXAMPLE PROGRAM TO READ A LIST OF STUDENTS INFORMATION AND PRINT
THEM IN ASCENDING ORDER OF THEIR AVERAGE MARKS USING ARRAY OF
STRUCTURES */
#include<stdio.h>
struct student
{
        char name[50],branch[5];
        int sub1,sub2,sub3;
        float avg;
}s[10];
main( )
{
        int i,j,n;
        struct student temp;
        clrscr();
        printf("\nEnter how many students:");
        scanf("%d",&n);
        for(i=1;i<=n;i++)
        {
          fflush(stdin);
          printf("\nEnter student %d Name:",i);
          gets(s[i].name);
          fflush(stdin);
          printf("\nEnter student %d Branch:",i);
          gets(s[i].branch);
          printf("\nEnter 3 subject Marks:");
          scanf("%d%d%d",&s[i].sub1,&s[i].sub2,&s[i].sub3);
        }
        for(i=1;i<=n;i++)
        s[i].avg=(s[i].sub1+s[i].sub2+s[i].sub3)/3.0;

        for(i=1;i<=n-1;i++)
        {
                for(j=i+1;j<=n;j++)
                {
                        if(s[i].avg>s[j].avg)
                        {
                                temp=s[i];
                                s[i]=s[j];
                                s[j]=temp;
                        }
                }
        }
        printf("\nStudent Details As Per Their Averge Marks Are:");
        for(i=1;i<=n;i++)
        printf("\n%s\t%s\t%.2f",s[i].name,s[i].branch,s[i].avg);
}
```

/* Write a program to display date month and year using structure*/

```
#include <stdio.h>
#include <string.h> /* We need the system-provided strcpy() */
void main( )
{
struct date /* declare the shape of date */
{
int day;
int year;
char month_name[10];
};
struct date d; /* create a variable d of type date */
d.day= 25;
strcpy(d.month_name,"January");
d.year= 2006; /* suppose positive is A.D. */
printf("Day= %d Month= %s Year= %d\n", d.day,d.month_name,d.year);
getch( );
}
```

OUTPUT:

```
Day= 25 Month= January Year= 2006
```

typedef :

The C programming language provides a keyword called **typedef.** By using **typedef** we can create new data type. The statement typedef is to be used while defining the new data type. The syntax is as given : ***typedef datatype dataname***

EG: typedef unsigned char BYTE;

After this type definitions, the identifier BYTE can be used as an abbreviation for the type **unsigned char, for example:** BYTE b1, b2;

By convention, uppercase letters are used for these definitions to remind the user that the type name is really a symbolic abbreviation, but you can use lowercase, as follows:

typedef unsigned char byte;

<u>typedef vs #define</u>

The **#define** is a C-directive which is also used to define the aliases for various data types similar to **typedef** but with some differences:

- The **typedef** is limited to giving symbolic names to types only where as **#define** can be used to define alias for values as well, like you can define 1 as ONE etc.
- The **typedef** interpretation is performed by the compiler where as **#define** statements are processed by the pre-processor.

/* WRITE A PROGRAM TO CREATE USER DEFINED DATATYPE HOURS ON INT DATA TYPE AND USE IT IN THE PROGRAM */

```
#include<stdio.h>
#include<conio.h>
void main( )
{
        typedef int hours;
        hours hrs;
        clrscr( );
        printf("Enter hours:");
        scanf("%d",&hrs);
        printf("\nMinutes  = %d",hrs*60);
        printf("\nSeconds = %d",hrs*60*60);
        getch( );

}
```

OUTPUT:

```
Enter hours:8
Minutes  = 480
Seconds = 28800
```

We can use **typedef** to give a name to user defined data type as well. For example we can use **typedef** with **structure** to define a new data type and then use that data type to define structure variables directly as follows:

```
#include <stdio.h>
#include <string.h>
typedef struct Books
{
    char  title[50];
    char  author[50];
    char  subject[100];
    int   book_id;
}Book;
void main( )
{
      Book b1;
      strcpy(b1.title, "C Programming");
      strcpy(b1.author, "knreddy");
      strcpy( b1.subject, "C Programming notes");
      b1.book_id = 6495407;
      clrscr( );
      printf( "Book title : %s\n",b1.title);
      printf( "Book author : %s\n",b1.author);
      printf( "Book subject : %s\n", b1.subject);
      printf( "Book book_id : %d\n", b1.book_id);
      getch( );
}

OUTPUT:
Book title : C Programming
Book author : knreddy
Book subject : C Programming notes
Book book_id : 6495407
```

/*CREATE USER DEFINED DATA TYPES FROM STRUCTURE. THE STRUCTURE SHOULD CONTAIN THE VARIABLES SUCH AS CHAR,INT etc. BY USING THESE VARIABLES DISPLAY NAME, GENDER AND ACCOUNTNUMBER OF TWO EMPLOYEES. USE ARRAY OF STRUCTURES.*/

```
#include<stdio.h>
#include<conio.h>
void main( )
{
    typedef struct{
            char name[20];
            char gender[8];
            int accno;
            } info;
    info employee[2];
    int k;
    clrscr( );
    for(k=0;k<2;k++)
    {
       printf("Name of the employee:");
       scanf("%s",employee[k].name);
       printf("gender   :");
       scanf("%s",employee[k].gender);
       printf("Account number   :");
       scanf("%d",&employee[k].accno);
    }
printf("\nNAME\tGENDER\tACCNO\n");
     for(k=0;k<2;k++)
     {
        printf("%s\t",employee[k].name);
        printf("%s\t",employee[k].gender);
        printf("%d\n",employee[k].accno);
     }
     getch( );
}
```

OUTPUT:

```
Name of the employee:knreddy
gender   :male
Account number   :387
Name of the employee:parinita
gender   :female
Account number   :388

NAME        GENDER      ACCNO
knreddy       male         387
parinita     female        388
```

Bit Fields

Bit-fields provide a mechanism that allows you to define variables that are each one or more binary bits within a single integer word, which you can nevertheless refer to explicitly with an individual member name for each one.

■**Note** Bit-fields are used most frequently when memory is at a premium and you’re in a situation in which you must use it as sparingly as possible. This is rarely the case these days so you won't see them very often. Bit-fields will slow your program down appreciably compared to using standard variable types. You must therefore assess each situation upon its merits to decide whether the memory savings offered by bit-fields are worth this price in execution speed for your programs. In most instances, bit-fields won’t be necessary or even desirable, but you need to know about them.

Suppose that you're programming an employee database program that keeps records on your company's employees. Many of the items of information that the database stores are of the yes/no variety, such as "Is the employee enrolled in the dental plan?" or "Did the employee graduate from college?" Each piece of yes/no information can be stored in a single bit, with 1 representing yes and 0 representing no.

Using C's standard data types, the smallest type you could use in a structure is a type char. You could indeed use a type char structure member to hold yes/no data, but seven of the char's eight bits would be wasted space. By using bit fields, you could store eight yes/no values in a single char.

Suppose your C program contains a number of TRUE/FALSE variables grouped in a structure called **status**, as follows:

```
struct
{           unsigned int widthValidated;
            unsigned int heightValidated;
} status;
```

This structure requires 8 bytes of memory space but in actual we are going to store either 0 or 1 in each of the variables. The C programming language offers a better way to utilize the memory space in such situation. If you are using such variables inside a structure then you can define the width of a variable which tells the C compiler that you are going to use only those numbers of bytes.

For example above structure can be re-written as follows:

```
struct{         unsigned int widthValidated : 1;
                unsigned int heightValidated : 1;
} status;
```

Now the above structure will require 4 bytes of memory space for status variable but only 2 bits will be used to store the values. If you will use upto 32 variables each one with a width of 1 bit, then also status structure will use 4 bytes, but as soon as you will have 33 variables then it will allocate next slot of the memory and it will start using 64 bytes. Let us check the following example to understand the concept:

```
#include <stdio.h>
#include <string.h>
#include<conio.h>
struct                          /* define simple structure */
{
  unsigned int widthValidated;
  unsigned int heightValidated;
} status1;
struct                          /* define a structure with bit fields */
{
  unsigned int widthValidated : 1;
  unsigned int heightValidated : 1;
} status2;
 void  main( )
{
   printf( "Memory size occupied by status1 : %d\n", sizeof(status1));
   printf( "Memory size occupied by status2 : %d\n", sizeof(status2));
   getch( );
}
```

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

```
Memory size occupied by status1 : 8
Memory size occupied by status2 : 4
```

Bit Field Declaration

The declaration of a bit-field has the form inside a structure:

```
struct{
  type  member_name : width ;
};
```

Below the description of variable elements of a bit field:

Elements	Description
type	An integer type that determines how the bit-field's value is interpreted. The type may be int, signed int, unsigned int.
member_name	The name of the bit-field.
width	The number of bits in the bit-field. The width must be less than or equal to the bit width of the specified type.

The variables defined with a predefined width are called **bit fields**. A bit field can hold more than a single bit for example if you need a variable to store a value from 0 to 7 only then you can define a bit field with a width of 3 bits as follows:

```
struct{
  unsigned int age : 3;
} Age;
```

The above structure definition instructs C compiler that age variable is going to use only 3 bits to store the value; if you will try to use more than 3 bits then it will not allow you to do so.

Let us try the following example:

```
#include <stdio.h>
#include <string.h>
struct{
  unsigned int age : 3;
} Age;
void main( )
{
   Age.age = 4;
   printf( "Sizeof( Age ) : %d\n", sizeof(Age) );
   printf( "Age.age : %d\n", Age.age );
   Age.age = 7;
   printf( "Age.age : %d\n", Age.age );
   Age.age = 8;
   printf( "Age.age : %d\n", Age.age );
   getch( );
}
```

OUTPUT:

```
Sizeof( Age ) : 4
Age.age : 4
Age.age : 7
Age.age : 0
```

Do we really gain that much by using bit fields?

Yes, you can gain quite a bit with bit fields. Consider a circumstance in which a file contains information from a survey. People are asked to answer TRUE or FALSE to the questions asked. If you ask 100 questions of 10,000 people and store each answer as a type char as T or F, you will need **10,000 * 100 bytes** of storage (because a character is 1 byte). This is 1 million bytes of storage. If you use bit fields instead and allocate one bit for each answer, you will need **10,000 * 100 bits.** Because 1 byte holds 8 bits, this amounts to 130,000 bytes of data, which is significantly less than 1 million bytes.

/*WRITE A PROGRAM TO DISPLAY THE EXAMINATION RESULT OF THE STUDENT USING BIT FIELDS.

```
#include<stdio.h>
#include<conio.h>
#define pass 1;
#define fail 0;
#define A 1;
#define B 2;
#define C 3;
void main( )
{
    struct student{
           char *name;
           unsigned result:1;
           unsigned grade:2;
           };
    struct student x;
    x.name="sachin";
    x.result=pass;
    x.grade=A;
    printf("\nResult= 1 means PASS\tResult=0 means FAIL");
    printf("\nGrade=1 means A grade\tGrade=2 means B grade\tGrade=3 means C grade");
    printf("\n\nName  :%s",x.name);
    printf("\nResult:%d",x.result);
    printf("\nGrade :%d",x.grade);
    getch( );
}
```

OUTPUT:

```
Result= 1 means PASS    Result=0 means FAIL
Grade=1 means A grade   Grade=2 means B grade   Grade=3 means C grade

Name  :sachin
Result:1
Grade :1
```

ENUMERATED DATA TYPES

An enumeration consists of a set of named integer constants. An enumeration type declaration gives the name of the (optional) enumeration tag and defines the set of named integer identifiers (called the "enumeration set," "enumerator constants," "enumerators," or "members"). A variable with enumeration type stores one of the values of the enumeration set defined by that type.

Variables of **enum** type can be used in indexing expressions and as operands of all arithmetic and relational operators. Enumerations provide an alternative to the **#define** preprocessor directive with the advantages that the values can be generated for you and obey normal scoping rules.

In ANSI C, the expressions that define the value of an enumerator constant always have **int** type; thus, the storage associated with an enumeration variable is the storage required for a single **int** value. An enumeration constant or a value of enumerated type can be used anywhere the C language permits an integer expression.

Enumerated types provide the facility to specify the possible values of a variable by meaningful symbolic names. The general format for defining an enumerated type is

enum tag { value1, value2, ..., valueN };

where tag is an identifier that names the enumerated type, and value1, value2, ..., valueN are identifiers, called enumerated constants. For example, the declaration

enum Days { Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday };

defines an enumerated types Days whose list of possible values are Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, and Saturday. Each of the values in the enumerated type has an ordinal value associated with it. The default values start at 0 and proceed in increments of 1 until all values in the list have an ordinal value. In the list of values for Days, the default ordinal values are Sunday has 0, Monday has 1, Tuesday has 2, Wednesday 3, Thursday has 4, Friday has 5 and Saturday has 6. An integer value may explicitly be associated with an enumeration constant by following it with = and a constant expression of integral type. Subsequent enumeration constants without explicit associations are assigned integer values one greater than the value associated with the previous enumerated constant. For example, the declaration

enum Days {Sunday=6,Monday, Tuesday, Wednesday=2, Thursday, Friday, and Saturday};

results in the value 6 being associated with Sunday , 7 with Monday , 8 with Tuesday, 2 with Wednesday , 3 with Thursday, 4 with Friday, and 5 with Saturday. Any signed integer value may be associated with an enumeration constant. The same integer value may be associated with two different enumerated constants in the same type declaration. An enumerated constant must be unique with respect to other enumerated constants and variables within the same name scope. Thus, in the presence of the declaration of days, the declaration

enum DaysOff { Saturday, Sunday };

is illegal because the identifiers Saturday and Sunday have already been defined to be enumerated constant values of days. Similarly, the variable declaration **float Monday;** following the declaration of Days, is also illegal.

Variables may be declared to be of an enumerated type in the same declaration containing the enumerated type definition, or in subsequent declarations of the form

enum tag variablelist;

Thus, weekday and caldays may be declared to be of type enum days

enum Days{ Sunday,Monday, Tuesday, Wednesday, Thursday, Friday, and Saturday} weekday, caldays; or **enum Days weekday, caldays;**

The tag may be omitted from the declaration of an enumerated type if all the variables of this type have been declared at the same time. Thus, the programmer may write

enum { Sunday,Monday, Tuesday, Wednesday, Thursday, Friday, and Saturday} weekday, caldays;

but due to the omission of the enumerated tag, the programmer may not subsequently declare another variable whose type is the same as that of weekday or caldays. A variable of a particular enumerated type can be assigned enumerated constants specified in the definition of the enumeration type. For example, the programmer may write **weekday = Tuesday;**

The programmer may also compare values as in **if (weekday == caldays)**

All enumerated types are treated as integer types, and the type of enumerated constants is treated as int. However, the programmer should differentiate between enumerated and integer types as a matter of good programming practice and use casts if they have to be mixed, as show in the following example:

typedef enum{Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday } DAYS;

The following rules apply to the members of an enumeration set:

- An enumeration set can contain duplicate constant values. For example, you could associate the value 0 with two different identifiers, perhaps named null and zero, in the same set.
- The identifiers in the enumeration list must be distinct from other identifiers in the same scope with the same visibility, including ordinary variable names and identifiers in other enumeration lists.
- Enumeration tags obey the normal scoping rules. They must be distinct from other enumeration, structure, and union tags with the same visibility.

```
/* EXAMPLE PROGRAM TO USE ENUMERATED DATA TYPE */
enum Days{Mon, Tue, Wed, Thu, Fri, Sat, Sun};
main( )
{
        enum Days start,end;
        clrscr();
        start=Tue;
        end=Sat;
        printf("\n%d   %d",start,end);
        start=65;
        printf("\n%d   %d",start,end);
}
```

It is also to associate numbers other than the sequence starting at zero with the names in the enum data type by including a specific initialization in the variable name list.

```
/* EXAMPLE PROGRAM FOR ENUM WITH DIFFERENT INITIALIZATIONS */
enum coins{p1,p2=45,p3,p4=80};
main( )
{
        clrscr();
        printf("\nValue of p1:%d",p1);
        printf("\nValue of p2:%d",p2);
        printf("\nValue of p3:%d",p3);
        printf("\nValue of p4:%d",p4);
}
```

Enumeration provides a convenient way to associate constant values with names as alternative to #define statements.

```
Example:     #define TRUE 1
             #define FALSE 0
Example:     enum flag{FALSE, TRUE};
```

PREPROCESSOR DIRECTIVES:

Any program execution needs certain steps. They are

a. The C program is written in the editor,
b. Compilation,
c. linking and
d. The executable code is generated.

In between these stages there also involves one more stage i.e. pre-processor. The preprocessor is a program that process the source program before it is passed on to the compiler.

The program typed in the editor is the source code to the pre-processor. The preprocessor then passes the source code to the compiler. It is also non essential to write the program with preprocessor facility. But it is a good practice to use it preferably at the beginning.

One of the most important features of C language is to offer preprocessor commands. The preprocessor commands are always initialized at the beginning of the program. It begins with symbol # (hash). It can be placed anywhere but quite often it is declared at the beginning before the main ( ) function or any particular function

The C Preprocessor provides several tools that are unavailable in other high level languages. The programmer can use these tools to make his program easy to read, easy to modify, portable and more efficient.

- The C Preprocessor, as it's name implies, is a program that processes the source code before it passes through the compiler. It operates under the control of what is known as **preprocessor command lines (or) directives.**
- Preprocessor directives are placed in the source program before the '**main'** function**.**
- Before the source code passes through the compiler, it is examined by the preprocessor for any preprocessor directives.
- Preprocessor directives follow special syntax rules that are different from the normal 'C' syntax rules. They are all begin with the symbol **'#'** (hash) in column one and do not require semi colon (;) at end.

A set of commonly used Preprocessor directives and their functions are shown below:

Preprocessor Directive	**Functionality**
#include	specified files to be included
#define	defines macro substitution
#undef	Undefines a preprocessor macro
#ifdef	Returns true if this macro is defined

#if	Tests if a compile time condition is true
#endif	Ends preprocessor conditional
#else	The alternative for #if
#ifndef	Returns true if this macro is not defined
#elif	#else an #if in one statement
#error	Prints error message on stderr
#pragma	Issues special commands to the compiler, using a standardized method

Classification of 'C' Preprocessor directives:

The 'C' Preprocessors directives can be divided into 3 categories:

i) **Macro Substitution directives**

ii) **File inclusion directives**

iii) **Conditional control directives**

MACRO SUBSTITUTION:

a) #define:

- It is a process where an identifier in a program is replaced by a predefined string composed of one or more tokens.
- The Preprocessor accomplishes this task under the direction of **#define** statement.
- This statement is usually known as a **macro definition** takes the following general format:

#define identifier string

If this statement is included in the program at the beginning, then the preprocessor replaces every occurrence of the identifier in the source code by the string.

- The keyword **#define** is written just as shown followed by the identifier and a string, with at least one blank space between them.
- The definition is not terminated by semicolon (;).

Eg: **#define PI 3.1415** where, PI → macro template

3.1415 → macro expansion

- Blanks or tabs are used to separate macro template & macro expansion.

Note: A space between the # and **define is optional.**

<u>A few examples are illustrated below for understanding macros</u>.

Eg 1: Use the identifier for 3.1413 as PI and write a program to find area of circle using it.

```
# include<stdio.h >
#include<conio.h>
#define PI 3.1413
void main( )
{
    float r, area;
    printf("\n Enter radius of circle in cms.");
    scanf("%f",&r);
    area=PI*r*r;
    printf("Area of a circle = %.2f cm2",area);
    getch( );
}
```

OUTPUT

```
Enter radius of circle in cms. : 7
Area of a circle = 153.86 cm2
```

<u>Eg 2:</u> Write a program to define and create identifier for C statements and variables

1) Read N as 10

2) Replace clrscr ( ) with cls

3) Replace getche( ) with wait( )

4) Replace pintf with display

```
#include<stdio.h>
#include<conio.h>
#define N 10
#define cls clrscr( )
#define wait( ) getche( )
#define display printf
void main( )
{
    int k;
    cls;
    for(k=1;k<=N;k++)
    display("%d\t",k);
    wait( );
}
```

OUTPUT

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

<u>Eg 3</u>: Write a program to define macros for logical operators.

```
#include<stdio.h>
#include<conio.h>
#define and &&
#define equal = =
#define larger >
void main( )
{
    int a,b,c;
    clrscr( );
    printf("Enter three numbers:");
    scanf("%d%d%d",&a,&b,&c);
    if(a larger b and a larger c)
            printf("%d is the larger number.", a);
   else if(b larger a and a larger c)
          printf("%d is the larger number.", b);
    else if(c larger a and c larger b)
          printf("%d is the larger number.", c);
    else if( a equal b and b equal c)
          printf("\n Numbers are same.");
    getch ( );
}
```

OUTPUT

```
 Enter three numbers: 5 4 7
 7 is the larger number.
```

<u>Eg 4:</u> Write a program to create identifier for displaying double and triple of a number.

```
#include<stdio.h>
#include<conio.h>
#define DOUBLE(a) a*2
#define TRIPLE(a) a*3
void main( )
{
    int a=1;
    clrscr( );
    printf("\n SINGLE\tDOUBLE\tTRIPLE");
    for(;a<=3;a++)
    printf("\n%d\t%d\t%d",a,DOUBLE(a),
                        TRIPLE(a));
    getch( );
}
```

OUTPUT:

```
SINGLE   DOUBLE   TRIPLE


    1         2           3
    2         4           6
    3         6           9
```

b) Undefining a macro

A macro can be undefined with **#undef** directive.

Syntax **#undef identifier**

It is useful when we do not want to allow the use of macros in any portion of the program.

Eg 1: Write a program to undefine a macro.

```
#include<stdio,h>
#include<conio.h>
#define wait getche( )
void main( )
{
   int k;
   #undef  wait getche( );
   clrscr( );
   for(k=1;k<=5;k++)
   printf("%d\t",k);
   wait;
}
```

EXPLANATION: In the above program wait() is defined in place of getche().In the program #undef directive undefines the same macro.Hence the compiler flags an error message "undefined symbol 'wait' in function main."

TOKEN PASTING AND STRINGIZING OPERATORS

Stringizing operation: In this operation macro argument is converted to string. The sign # carries the operation. It is placed before the argument.

Eg 1: Write a program to carry out stringizing operation.

```
#include<stdio.h>
#include<conio.h>
#define say(m) printf(#m)
void main()
{
   clrscr();
   say(Hello);
}
```

```
OUTPUT:      Hello
```

EXPLANATION: In the above program after conversion the statement say(Hello) is treated as printf("Hello"). It is not essential to enclose the text in quotation marks in the stringizing operator.

Eg 2: Write a program to carry stringizing operation and macro arguments.

```
#include<stdio.h>
#include<conio.h>
#define DOUBLE(x) printf(“Double of “#x” =%d\n”,x*2)
#define TRIPLE(x)   printf(“Triple of “#x” =%d\n”,x*3)
void main( )
{
   int m;
   clrscr( );
   printf(“Enter a number:”);
   scanf(“%d”,&m);
   DOUBLE(m);
   TRIPLE(m);
   getche( );
}
```

OUTPUT

```
Enter a number: 5
Double of m =  10
Triple of m = 15
```

EXPLANATION: In the above program the value of ‘m’ is passed to DOUBLE( ) and TRIPLE( ) macros which is assigned to x. ”#x” prints the name of the variable passed through the macros.

Eg 3: Write a program to find the larger of two numbers using macro with arguments.

```
#include<stdio.h>
#include<conio.h>
#define MAX(x,y)  if(x>y) c=x; else c=y;
void main( )
{
  int  x=5,y=8,c;
 clrscr( );
 MAX(x,y);
 printf(“\n Larger of two numbers=%d”,c);
}
```

OUTPUT

```
Larger of two numbers=8
```

EXPLANATION: In the above program macro MAX( ) is defined with two arguments x and y. When a macro is called, its corresponding expression is executed and result is displayed .The expression contains if statement that determines the largest number and assign it to variable c.

FILE INCLUSION DIRECTIVES:

a) The #include Directive

The #include directive loads specified file in the current program. The macros and functions of loaded file can be called in the current program. The included file is also compiled with the current program. The syntax is as given below.

a) #include “filename”

b) #include <filename>

Where **‘#’ is a symbol used with directives.**

a) The file name is included in the double quotation marks which indicates that the search for the file is made in the current directory and in the standard directories.

Example: #include “stdio.h”

b) When the file name is included without double quotation marks, the search for file is made only in standard directories.

Example: #include<stdio.h>
#include<udf.h>

Eg 1: Write a program to call the function undefined in “udf.c”.

```
#include<stdio.h>
#include<conio.h>
#include”udf.c”
void main( )
{
    clrscr( );
    display( );
}
```

OUTPUT

```
Function called
```

Contents of udf.c file

```
int display();display( ) {printf”\n Function called”);return 0;}
```

Explanation: In the first program the “udf.c” is included. It is a user defined function file. It is compiled before the main program is compiled. The complete programs along with the included one are executed. The output of the program “Function Called” is displayed.

CONDITIONAL COMPILATION DIRECTIVES:

The most frequently used conditional compilation directives are 3. They are **#if, #else, #endif** etc. These directives allow the programmer to include the portions of the codes based on the conditions. The compiler compiles selected portion of the source codes based on the conditions.

a) '# ifdef' directive: The syntax of the **#ifdef** directives is given below.

Syntax

`#ifdef<identifier>`
`{`
`statement1;`
`statement2;`
`}`
`#else`
`{`
`statement3;`
`statement4;`
`}`
`#endi`
	The #ifdef preprocessor tests whether the identifier has defined substitute text or not. If the identifier is defined then #if block is compiled and executed. The compiler ignores #else block even if errors are intentionally made. Error messages will not be displayed .If the identifier is not defined then #else block is compiled and executed.

Eg 1: Write a program to use conditional compilation statement as to whether the identifier is defined or not.

```
#include<stdio.h>
#include<conio.h>
#define LINE   1
void main( )
{
  clrscr( );
  #ifdef LINE
      printf("This is line number one");
  #else
      printf("This is line number two");
  #endif
  getche( );
}
OUTPUT:     This is line number one.
```

Explanation:

In the above program #ifdef checks whether the LINE identifier is defined or not. If defined the #if block is executed .On execution the output of the program is" This is line number one". In case the identifier is undefined the #else block is executed and output is" This is line number two".

<u>Eg 2:</u> Write a program similar to above with conditional compilation directives as to whether the identifier is defined or not.

```
#include<stdio.h>
#include<conio.h>
#define  E =
void main()
{
   int a,b,c,d;
   clrscr( );

   #ifdef E
    {
       a E 2;
       b E 3;
       printf("A=%d & B=%d",a,b);
    }
  #else
   {
     c=2;
     d=3;
     printf("C=%d & D=%d",c,d);
   }
#endif
getche( );
}
```

OUTPUT

```
 A=2 & B=3
```

Explanation:

The execution of the above program is same as the previous one. The only difference is in the name of the identifier .Here, in this program the identifier is E. The compiler searches identifier E. If it is found then execution of #if block takes place otherwise #else block. The #endif statement indicates the end of #if #endif block.

<u>The #ifndef Directive</u>:

The syntax of the #ifndef directive is given below.

Syntax

```
#ifndef<identifier>
{
     statement1;
     statement2;
}
#else
{
     statement3;
     statement4;
}
#endif
```

The #ifndef works exactly opposite to that of #ifdef. The #ifndef preprocessor tests whether the identifier has defined substitute text or not. If the identifier is defined then #else block is compiled and executed and the compiler ignores #if block even if errors are intentionally made. Error messages will not be displayed .If identifier is not defined then #if block is compiled and executed.

<u>Eg 1</u>: Write a program to check conditional compilation directives '#ifndef'. If it is observed display one message otherwise another message.

```
#include<stdio.h>
#include<conio.h>
#define T  8
void main( )
{
    clrscr( );
    #ifndef  T
        printf("\n Macro is not defined.");
    #else
       printf("\n Macro is defined.");
    #endif
}
```

OUTPUT

```
 Macro is defined.
```

Explanation:
In the above program #ifndef checks for the identifier T. If it is defined the #else block is executed. On the execution of the block the output of the program is "Macro is defined". In case the identifier is undefined the #else block is executed and output is" Macro is not defined."

The #error Directive

The #error directive is used to display user defined message during compilation of the program. The syntax is as given below.

```
#if  !defined  (identifier)
#error <ERROR MESSAGE>
#endif
```

Eg 1: Write a program to display user-defined error message using #error directive.
#include<stdio.h>
#include<conio.h>
#define B 1
void main( )
{
   clrscr( );
   #if !defined(A)
      #error MACRO A IS NOT DEFINED
   #else
   printf(“Macro found.”);
   #endif
}
	Explanation:
	In the above program identifier ‘B’ is defined. In the above absence of identifier an error is generated and the # error directive displays the error message. The error message is user defined and displayed in the message box at the bottom of the editor.

Tip: The #defined directive will work exactly opposite to **#! defined** directive. The syntax is given below.

FILES

Any data that the user enters when the program is executed is lost once the program finishes running. At the moment, if the user wants to run the program with the same data, he or she must enter it again each time. There are a lot of occasions when this is not only inconvenient, but also makes the programming task impossible.

If you want to maintain a directory of names, addresses, and telephone numbers, for instance, a program in which you have to enter all the names, addresses, and telephone numbers each time you run it is worse than useless! The answer is to store data on permanent storage that continues to be maintained after your computer is switched off. This storage is called a **file**, and a file is usually stored on a hard disk.

"File is a set of records that can be accessed through the set of library functions."

FILE TYPES:

There are two types of files: 1) Sequential files 2) Random accesses file.

A file that is organized as a stream, with writing allowed only at the end, is called a sequential file. If the underlying type of data in the stream is character, it is further known as a text file.

A file that may have any of the data elements read or written directly by some indexing scheme without having to start at the beginning is called a random-access file.

STEPS FOR FILE OPERATIONS

Accessing Files

The files that are resident on your disk drive each have a name, and the rules for naming files will be determined by your operating system.

When you process a file in C, your program references a file through a **file pointer**. A file pointer is an abstract pointer that is associated with a particular file when the program is run so that the program can work with different files on different occasions. A file pointer points to a struct that represents a stream.

If you want to use several files simultaneously in a program, you need a separate file pointer for each file, although as soon as you've finished using one file, you can associate the file pointer you were using with another file. So if you need to process several files, but you'll be working with them one at a time, you can do it with one file pointer.

There are different operations that can be carried out on a file. These are:

(a) Creation of a new file

(b) Opening an existing file

(c) Reading from a file

(d) Writing to a file

(e) Moving to a specific location in a file (seeking)

(f) Closing a file

Commonly Used C File-System Functions

FUNCTION	OPERATION
fopen( )	Opens a file
fclose( )	Closes a file
putc( )	Writes a character to a file
fputc( )	Same as putc( )
getc( )	Reads a character from a file
fgetc( )	Same as getc( )
fgets( )	Reads a string from a file
fputs( )	Writes a string to a file
fseek( )	Seeks to a specified byte in a file
ftell( )	Returns the current file position
fprintf( )	Is to a file what printf( ) is to the console
fscanf( )	Is to a file what scanf( ) is to the console
feof( )	Returns true if end-of-file is reached
ferror( )	Returns true if an error has occurred
rewind( )	Resets the file position indicator to the beginning of the file
remove( )	Erases a file
fflush( )	Flushes a file
rename( )	change the name of the file

OPENING A FILE:

We can associate a specific external file name with an internal file pointer variable through a process referred to as **opening a file**. We can open a file by calling the standard library function fopen( ), which returns the file pointer for a specific external file. The function fopen( ) is defined in **<stdio.h>,** and it has this prototype: **FILE *fopen(char *name, char *mode);**

The first argument to the function is a pointer to a string that is the name of the external file that you want to process. You can specify the name explicitly as an argument, or you can use an array, or a variable of type pointer to char that contains the address of the character string that defines the file name. The second argument to the fopen( )function is a character string called the **file mode** that specifies what you want to do with the file.

■***Note*** *Notice that a file mode specification is a character string between double quotes, not a single character between single quotes.*

The **fopen( )** performs three important tasks when you open the file :

a) Firstly it searches on the disk the file to be opened.

b) a) In case file exists, then it loads the file from the disk into a place in memory called buffer.

b) If the file does not exist this function returns a NULL. NULL is a macro defined character in the header file <stdio.h>. This indicates that it is unable to open file. There may be following reasons for failure of fopen( ) function.

i) When the file is in protected or hidden mode.

ii) The file may be used by another program.

c) It sets up a character pointer that points to the first character of the buffer. Whenever a file is opened the character pointer points to the first character of the file.

<u>Legal Values for Mode</u>

Mode	Meaning
r	Open a text file for reading
w	Create a text file for writing
a	Append to a text file
rb	Open a binary file for reading
wb	Create a binary file for writing
ab	Append to a binary file
r+	Open a text file for read/write
w+	Create a text file for read/write
a+	Append or create a text file for read/write
r+b	Open a binary file for read/write
w+b	Create a binary file for read/write
a+b	Append or create a binary file for read/write

CLOSING A FILE

When you've finished with a file, you need to tell the operating system that this is the case and free up your file pointer. This is referred to as **closing** a file. This can be done by calling the fclose( ) function which accepts a file pointer as an argument and returns a value of type int, which will be EOF if an error occurs and 0 otherwise. The typical usage of the fclose( ) function is as follows:

fclose(fp); /* Close the file associated with pfile */

The result of executing this statement is that the connection between the pointer, fp, and the physical file name is broken, so fp can no longer be used to access the physical file it represented. If the file was being written, the current contents of the output buffer are written to the file to ensure that data isn't lost.

■***Note*** *EOF is a special character called the* ***end-of-file character****. In fact, the symbol EOF is defined in <stdio.h> and is usually equivalent to the value –1. However, this isn't necessarily always the case, so you should use EOF in your programs rather than an explicit value. EOF generally indicates that no more data is available from a stream.*

It's good programming practice to close a file as soon as you've finished with it. This protects against output data loss, which could occur if an error in another part of your program caused the execution to be stopped in an abnormal fashion. This could result in the contents of the output buffer being lost, as the file wouldn't be closed properly. We must also close a file before attempting to rename it or remove it.

■***Note*** *Another reason for closing files as soon as you've finished with them is that the operating system will usually limit the number of files you may have open at one time. Closing files as soon as you've finished with them minimizes the chances of you falling afoul of the operating system in this respect.*

There is a function in <stdio.h> that will force any unwritten data left in a buffer to be written to a file. This is the function fflush( ), which you've already used in previous chapters to flush the input buffer. With your file pointer fp, you could force any data left in the output buffer to be written to the file by using this statement: **fflush(fp);**

The fflush( ) function returns a value of type int, which is normally 0 but will be set to EOF if an error occurs.

DELETING A FILE

Because you have the ability to create a file in your code, at some point you'll want to be able to delete a file programmatically, too. The remove( ) function that's declared in <stdio.h> does this. **remove("filename.txt");**

This will delete the file from the current directory that has the name filename.txt. Note that the file should not be open when you call remove( ) to delete it. If the file is open, the effect of calling remove is implementation-defined, so consult your library documentation.

/*WRITE A PROGRAM TO WRITE DATA TO TEXT FILE AND READ IT*/

```
#include<stdio.h>
#include<conio.h>
void main( )
{
    FILE *fp;
    char c=' ';
    clrscr( );
    fp=fopen("data.txt","w");
    if(fp==NULL)
    {
      printf("Cannot open a file");
      goto end;
    }
    printf("write data and press . to stop\n");
    while(c!='.')
    {
      c=getche( );
      fputc(c,fp);      //fprintf(fp,"%c",c);
    }
    fclose(fp);
    printf("\n read the contents");
    fp=fopen("data.txt","r");
    while(!feof(fp))
    printf("%c",getc(fp));
    end:
    getch( );
}
```

In the above example the file data.txt is opened in write mode. If the data.txt file does not exist a new file is created as the file is opened in write mode. If the file exists then the previous data in the file is overwritten.

After writing to the file, the file is closed and again the same file is opened in the read mode.

/* WRITE A PROGRAM TO OPEN A PRE-EXISTING FILE AND ADD INFORMATION AT THE END OF FILE. DISPLAY THE CONTENTS OF THE FILE BEFORE AND AFTER APPENDING.*/

```
#include<stdio.h>
#include<conio.h>
void main( )
{
    FILE *fp;
    char c;
    clrscr( );
    printf("content of file before appending:\n");
    fp=fopen("data.txt","r");
    while(!feof(fp))
    {       c=fgetc(fp);
            printf("%c",c);
    }
    fp=fopen("data.txt","a");
    if(fp==NULL)
    {    printf("File file not exists");
         goto end;
    }
    printf("Enter string to append and press $ to end the string:");
    while(c!='$')
    {
      c=getche( );
      fputc(c,fp);
    }
    fclose(fp);
    printf("\nContents of file after appending:");
    fp=fopen("data.txt","r");
    while(!feof(fp))
    {
      c=fgetc(fp);
      printf("%c",c);
    }
    end:
    getch( );
}
```

OUTPUT:

```
content of file before appending:
KKKKKKK.nnnnnn$
Enter string to append and press $ to end the string: rrrrrr$
Contents of file after appending: KKKKKKK.nnnnnn$rrrrrr$
```

```
/*WRITE A PROGRAM TO OPEN A FILE FOR READ/WRITE OPERATIONS IN BINARY
MODE. READ AND WRITE NEW INFORMATION IN THE FILE.*/
#include<stdio.h>
#include<conio.h>
void main( )
{
    FILE *fp;
    char c=' ';
    clrscr( );
    fp=fopen("data1.dat","wb");
    if(fp==NULL)
    {
      printf("Cannot open a file");
      goto end;
    }
    printf("write data and press . to stop\n");
    while(c!='.')
    {
      c=getche();
      fputc(c,fp);
    }
    fclose(fp);
    printf("\n read the contents");
    fp=fopen("data1.dat","rb");
    while(!feof(fp))
    printf("%c",getc(fp));
    end:
    getch( );
}
```

FILE I/O:

1. **fprintf( ):** It is sometimes useful also to output to different streams. fprintf( ) allows us to do exactly that. This function is used for writing to a file.

The prototype for fprintf is: **int fprintf(FILE *stream, const char *format, ...);**

fprintf takes in a special pointer called a file pointer, signified by FILE *. It then accepts a formatting string and arguments. The only difference between fprintf and printf is that fprintf can redirect output to a particular stream. These streams can be stdout, stderr, or a file pointer.

2. **fscanf ( ):** This function reads characters, strings, integers , floats from the file pointed by a file pointer.

The prototype for fscanf is**:** **int fscanf(FILE *stream, const char *format, ...);**

3. **getc( ) and fgetc( ):**

fgetc and getc both read a character from a file (input stream) and have the same syntax. They are for all intents and purposes the same!

Prototype: int fgetc(FILE *stream);

```
Syntax:         FILE *file_pointer;
                char ch;
                ch = fgetc(file_pointer);
                ch = getc(file_pointer);
```

4. **putc( ) and fputc( ):**

putc and fputc writes one character to a file.

Prototype: int fputc(int c, FILE *stream);

```
Syntax:         FILE *fp;
                    int  ch = 'a';
                    fp = fopen("/tmp/file", "w");
                    fputc(ch, fp);
                    fclose(fp);
```

5. **fgets( ) and fputs( ):**

These are useful for reading and writing entire lines of data to/from a file. If *buffer* is a pointer to a character array and *n* is the maximum number of characters to be stored, then

fgets (buffer, n, input_file);

will read an entire line of text (max chars = n) into *buffer* until the newline character or n=max, whichever occurs first. The function places a NULL character after the last character in the buffer. The function will be equal to a NULL if no more data exists.

fputs (buffer, output_file); writes the characters in *buffer* until a NULL is found. The NULL character is not written to the *output_file*.

Notes

- fgets should be used in preference to gets as it checks that the incoming data does not exceed the buffer size.
- If fgets is reading STDIN, the NEWLINE character is placed into the buffer. gets removes the NEWLINE.

6. putw( ) and getw( ):

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

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

```
/*  Write a program to write data into a file and read the data from the file. Use fprintf( ) and  fscanf( )   */
#include<stdio.h>
#include<conio.h>
void main( )
{
    FILE *fp;
    char string[30];
    fp=fopen("file1.txt","w");
    clrscr( );
    printf("\nenter text here: ");
    gets(string);
    fprintf(fp,"%s",string);
    fclose(fp);
    fp=fopen("file1.txt","r");
    printf("\ntest in the file is :");
    fscanf(fp,"%s",string);
    printf("%s",string);
    fclose(fp);
    getch( );
}

OUTPUT:
enter text here: knreddy
 test in the file is :knreddy
```

```
/*Write a program to enter integers and write them in the file using putw( ) and read integers
from the file using getw( )*/
#include<stdio.h>
#include<conio.h>
void main( )
{
    FILE *fp;
    int n;
    fp=fopen("num.txt","w");
     clrscr( );
    printf("\nenter numbers here.press 0 to stop: ");
    while(n!=0)
    {
       scanf("%d",&n);
       putw(n,fp);
    }
    fclose(fp);
    fp=fopen("num.txt","r");
    printf("\nnumbers in the file are :");
    while((n=getw(fp))!=EOF)
    {
       printf("%4d",n);
    }
    fclose(fp);
    getch( );
}
OUTPUT:
enter numbers here.press 0 to stop: 1    2    3    4    5    6    7    8    9    0
numbers in the file are :   1   2   3   4   5   6   7   8   9   0
```

STRUCTURES READ AND WRITE

The only function that is available for storing numbers in a disk file is the **fprintf( )** function. It is important to understand how numerical data is stored on the disk by **fprintf( ).** Text and characters are stored one character per byte, as we would expect. Are numbers stored as they are in memory, two bytes for an integer, four bytes for a float, and so on? No.

Numbers are stored as strings of characters. Thus, 1234, even though it occupies two bytes in memory, when transferred to the disk using **fprintf( )**, would occupy four bytes, one byte per character. Similarly, the floating-point number 1234.56 would occupy 7 bytes on disk. Thus, numbers with more digits would require more disk space.

Hence if large amount of numerical data is to be stored in a disk file, using text mode may turn out to be inefficient. The solution is to open the file in binary mode and use those functions (**fread( )** and **fwrite( )**) which store the numbers in binary format. It means each number would occupy same number of bytes on disk as it occupies in memory.

The fwrite( ) Function

The fwrite( ) library function writes a block of data from memory to a binary-mode file. Its prototype in STDIO.H is **int fwrite(void ****buf***, int** ***size*****, int** ***count*****, FILE *fp);**

The argument *buf* is a pointer to the region of memory holding the data to be written to the file. The pointer type is void; it can be a pointer to anything.

The argument *size* specifies the size, in bytes, of the individual data items, and *count* specifies the number of items to be written. For example, if you wanted to save a 100-element integer array, *size* would be 2 (because each int occupies 2 bytes) and *count* would be 100 (because the array contains 100 elements). To obtain the *size* argument, you can use the sizeof( ) operator.

The argument fp is, of course, the pointer to type FILE, returned by fopen( ) when the file was opened. The fwrite() function returns the number of items written on success; if the value returned is less than count, it means that an error has occurred.

The fread( ) Function

The fread() library function reads a block of data from a binary-mode file into memory. Its prototype in STDIO.H is **int fread(void ****buf***, int** ***size*****, int** ***count*****, FILE *fp);**

The argument *buf* is a pointer to the region of memory that receives the data read from the file. As with fwrite( ), the pointer type is void.

The argument *size* specifies the size, in bytes, of the individual data items being read, and *count* specifies the number of items to read. Note how these arguments parallel the arguments used by fwrite( ). Again, the sizeof( ) operator is typically used to provide the *size* argument. The argument fp is (as always) the pointer to type FILE that was returned by fopen( ) when the file was opened. The fread( ) function returns the number of items read; this can be less than *count* if end-of-file was reached or an error occurred.

```
/* WRITE A PROGRAM TO WRITE A BLOCK OF STRUCTURE ELEMENTS TO THE
GIVEN FILE USING FWRITE( ) FUNCTION*/
#include<stdio.h>
#include<conio.h>
void main( )
{
      struct{     char name[20];
                   int age;
             }stud;
      FILE *fp;
      int i;
      fp=fopen("stud.dat","wb");
      if(fp==NULL)
      {
        printf("File cannot open\n");
        goto end;
      }
       else
       {
         printf("\nENTER NAME:");
         scanf("%s",stud.name);
         printf("\nENTER AGE:");
         scanf("%d",&stud.age);
         fwrite(&stud,sizeof(stud),1,fp);
       }
       fclose(fp);
       end:
       getch( );
       }
```

```
/* WRITE A PROGRAM TO READ A BLOCK OF STRUCTURE ELEMENTS FROM THE
GIVEN FILE USING FREAD( ) FUNCTION*/
#include<stdio.h>
#include<conio.h>
void main( )
{
    struct{
        char name[20];
        int age;
        }stud;
    FILE *fp;
    int i,j=0,n;
    fp=fopen("stud.dat","rb");
    if(fp==NULL)
    {
     printf("File cannot open\n");
     goto end;
    }
    else
    {
      fread(&stud,sizeof(stud),1,fp);
      printf("\nname of stud:%s",stud.name);
      printf("\nage of stud: %d",stud.age);
     }
    fclose(fp);
    end:
    getch( );
}
```

OTHER FILE FUNCTION:

The fseek( ) Function:

int fseek(FILE *pfile, long offset, int origin);

The **fseek( )** function sets the file position indicator associated with *stream* according to the values of *offset* and *origin*. The first parameter is a pointer to the file that you're repositioning. The second and third parameters define where you want to go in the file. The second parameter is an offset from a reference point specified by the third parameter.Its purpose is to support random access I/O operations. The *offset* is the number of bytes from *origin* to seek to. The values for *origin* must be one of these macros (defined in **<stdio.h>**):

Name	**Meaning**
SEEK_SET	Seek from start of file
SEEK_CUR	Seek from current location
SEEK_END	Seek from end of file

A return value of zero means that **fseek( )** succeeded. A nonzero value indicates failure.

feof( ):

int feof(FILE *_stream_);

The **feof( )** function determines whether the end of the file associated with *stream* has been reached.

A nonzero value is returned if the file position indicator is at the end of the file; zero is returned otherwise.

Once the end of the file has been reached, subsequent read operations will return **EOF** until either

rewind( ) is called or the file position indicator is moved using **fseek( )**.

The **feof( )** function is particularly useful when working with binary files because the end-of-file marker is also a valid binary integer. Explicit calls must be made to **feof( )** rather than simply testing the return value of **getc( )**, for example, to determine when the end of a binary file has been reached.

ftell( ):

prototype is **long ftell(FILE *pfile);**

This function accepts a file pointer as an argument and returns a long integer value that specifies the current position in the file. This could be used with the file that's referenced by the pointer pfile that you've used previously, as in the following statement:

fpos = ftell(pfile);

The fpos variable of type long now holds the current position in the file and, as you'll see, you can use this in a function call to return to this position at any subsequent time. The value is actually the offset in bytes from the beginning of the file.

rewind( ):

rewind(pfile); This statement calls the rewind( ) function, which simply moves the current position back to the beginning of the file so that you can read it again

rename( )

int rename(const char ****oldfname*, const char ****newfname***);**

The **rename( )** function changes the name of the file specified by *oldfname* to *newfname*. The *newfname* must not match any existing directory entry. The **rename( )** function returns zero if successful and nonzero if an error has occurred.

```
/* WRITE A PROGRAM TO DEMONSTRATE FSEEK( )  */
#include<stdio.h>
#include<conio.h>
void main( )
{
    FILE *fp;
    int n, ch;
    char string[40];
    clrscr( );
    fp=fopen("text.txt","w");
    printf("\nEnter the text: ");
    gets(string);
    fputs(string,fp);
    fclose(fp);
    fp=fopen("text.txt","r");
    printf("\ncontents of file :");
    while(!feof(fp))
    printf("%c",getc(fp));
    printf("\nHow many characters including spaces would you like to skip: ");
    scanf("%d",&n);
    fseek(fp,n,SEEK_SET);
    printf("\n Information after %d bytes \n",n);
    while(!feof(fp))
    printf("%c",getc(fp));
    fclose(fp);
    getch( );
}
OUTPUT:
Enter the text: this is a c program for demonstration of fseek function
contents of file :this is a c program for demonstration of fseek function
How many characters including spaces would you like to skip10
Information after 10 bytes
c program for demonstration of fseek function
```

```
/* WRITE A PROGRAM TO DETECT THE END OF FILE USING FEOF( )*/
#include<stdio.h>
#include<conio.h>
main( )
{
    FILE *fp;
    char c;
    fp=fopen("text.txt","r");
    c=feof(fp);
    clrscr( );
    printf("\nfile pointer at the beginning of file :%d \n",c);
    while(!feof(fp))
    printf("%c",getc(fp));
    c=feof(fp);
    printf("\nfile pointer at the end of file:%d \n",c);
    fclose(fp);
    getch( );
}
```

OUTPUT:

```
file pointer at the beginning of file :0
this is a c program for demonstration of fseek function
file pointer at the end of file:16
```

Explanation: feof( ) function returns a non-integer value when end of the file is reached. In this program it had returned 16.

```
/* Write a program to print the current position of the file pointer in the file using ftell( ) */
#include<stdio.h>
#include<conio.h>
void main( )
{
    FILE *fp;
    int n, ch;
    char string[40];
    clrscr( );
    fp=fopen("text.txt","r");
    printf("\ncontents of file :");
    while(!feof(fp))
    printf("%c",getc(fp));
    printf("\nHow many characters including spaces would you like to skip: ");
    scanf("%d",&n);
    fseek(fp,n,SEEK_SET);
    printf("\n file pointer position after %d bytes \n",n);
    while(!feof(fp))
    printf("%c-%d\t",getc(fp),ftell(fp));
    fclose(fp);
    getch( );
}
```

OUTPUT:

```
contents of file :this is a c program, we are writing programs on files
How many characters including spaces would you like to skip: 40
file pointer position after 40 bytes
r-40   a-41   m-42   s-43    -44   o-45   n-46    -47   f-48   i-49
l-50   e-51   s-52    -53
```

Explanation: The file test.txt already contains some text.In this program first the file pointer is moving to some position using fseek( ) function. From there we are printing the position of file pointer and what is the character at that position.

```
/* WRITE A PROGRAM TO COPY THE CONTENTS OF ONE FILE TO OTHER FILE */
#include<stdio.h>
#include<conio.h>
int filecopy( char *oldname, char *newname );
void main( )
{
    char source[80], destination[80];
    /* Get the source and destination names. */
    printf("\nEnter source file: ");
    gets(source);
    printf("\nEnter destination file: ");
    gets(destination);
    if ( filecopy( source, destination ) ==0 )
      puts("Copy operation successful");
    else
      fprintf(stderr, "Error during copy operation");
    getch( );
}
int filecopy( char *oldname, char *newname )
{
   FILE *fold, *fnew;
   int c;
   /* Open the source file for reading in binary mode. */
   if ( ( fold = fopen( oldname, "rb" ) ) == NULL )
   return -1;
 /* Open the destination file for writing in binary mode. */
   if ( ( fnew = fopen( newname, "wb" ) ) == NULL )
   {
     fclose ( fold );
     return -1;
   }
/* Read one byte at a time from the source; if end of file */
/* has not been reached, write the byte to the destination. */
  while (1)
  {
      c = fgetc( fold );
      if ( !feof( fold ) )
        fputc( c, fnew );
      else
        break;
  }
  fclose ( fnew );
  fclose ( fold );
  return 0;
}
```

OUTPUT:

```
Enter source file: source.txt
Enter destination file: dest.txt
Copy operation successful
```

Explanation: In the above program it is considered that there are two existing files with the name source.txt and dest.txt. This program copies the contents present in source.txt file to dest.txt file.

WRITE A PROGRAM TO MERGE TWO FILES INTO A THIRD FILE */

```
#include<stdio.h>
#include<conio.h>
main( )
{
        char ch;
        FILE *f1,*f2,*f3;
        clrscr( );
        f1=fopen("demo1.txt","r");
        f3=fopen("result.txt","w");
        if(f1==NULL)
        {
            printf("\nFile Opening Error");
            exit();
        }
        while((ch=fgetc(f1))!=EOF)
        fputc(ch,f3);
        fcloseall();
        f2=fopen("demo2.txt","r");
        f3=fopen("result.txt","a");
        if(f2==NULL)
        {
            printf("\nFile Opening Error");
            exit( );
        }
        while((ch=fgetc(f2))!=EOF)
        fputc(ch,f3);
        fcloseall( );
        f3=fopen("result.txt","r");
        if(f3==NULL)
        {
            printf("\nFile Opening Error");
            exit();
        }
        printf("\nMERGING CONTENTS ARE:\n");
        while((ch=fgetc(f3))!=EOF)
        printf("%c",ch);
        fclose(f3);
}
```

```
/* WRITE A PROGRAM TO READ THE DATA FROM A FILE AND PRINT NUMBER OF
CHARACTERS, WORDS, DIGITS, LINES AND SPECIAL SYMBOLS */
#include<stdio.h>
#include<conio.h>
main( )
{
                char ch;
                FILE *fp;
                int nc=0,nd=0,nw=1,nl=1,nsp=0;
                clrscr();
                fp=fopen("check.txt","r");
                if(fp==NULL)
                {
                   printf("\nFILE OPENING ERROR");
                   exit();
                }
                while((ch=getc(fp))!=EOF)
                {
                   if((ch>='a'&&ch<='z') || (ch>='A'&&ch<='Z'))
                         nc=nc+1;
                   else if(ch>='0'&&ch<='9')
                         nd=nd+1;
                   else if(ch==' '||ch=='\t')
                         nw=nw+1;
                   else if(ch=='\n')
                   {
                         nw=nw+1;
                         nl=nl+1;
                   }
                   else
                   nsp=nsp+1;
                }
                printf("\n No of Characters:%d",nc);
                printf("\n No of Digits:%d",nd);
                printf("\n No of Words:%d",nw);
                printf("\n No of Lines:%d",nl);
                printf("\n No of Special Symbols:%d",nsp);
                fclose(fp);
}
```

```
/* WRITE A PROGRAM TO CREATE AN INPUT DATA FILE WHICH CONTAINS A LIST
OF INTEGERS.  READ THE SAME DATA FROM THE FILE AND WRITE EVEN AND
ODD NUMBERS INTO TWO SEPARTE FILES */
#include<stdio.h>
main( )
{
            int item,n,i;
            FILE *f1,*f2,*f3;
            clrscr();
            f1=fopen("input.dat","w");
            printf("\nEnter how many numbers:");
            scanf("%d",&n);
            printf("\nEnter %d Numbers:",n);
            for(i=1;i<=n;i++)
            {
              scanf("%d",&item);
              putw(item,f1);
            }
            fclose(f1);
            f1=fopen("input.dat","r");
            if(f1==NULL)
            {
              printf("\nFILE OPENING ERROR");
              exit();
            }
            f2=fopen("even.dat","w");
            f3=fopen("odd.dat","w");
            while((item=getw(f1))!=EOF)
            {
              if(item%2==0)
                   putw(item,f2);
              else
                   putw(item,f3);
            }
            fcloseall();
            printf("\nEVEN NUMBERS ARE:");
            f2=fopen("even.dat","r");
            while((item=getw(f2))!=EOF)
            printf("%6d",item);
            fclose(f2);
            printf("\nODD NUMBERS ARE:");
            f3=fopen("odd.dat","r");
            while((item=getw(f3))!=EOF)
            printf("%6d",item);
            fclose(f3);
}
```

```
/* WRITE A PROGRAM TO READ DATA FROM  A FILE AND PRINT POSITION BYTE
OF EACH CHARACTER IN THE FILE */
#include<stdio.h>
main( )
{
            FILE *fp;
            long n;
            char ch;
            clrscr();
            fp=fopen("win.dat","r");
            if(fp==NULL)
            {
              printf("\nFile Opening Error");
              exit();
            }
            while(1)
            {
              n=ftell(fp);
              if((ch=fgetc(fp))==EOF)
              exit();
              printf("\n%c position is: %ld Byte",ch,n);
            }
            fclose(fp);
}
```

```
/* WRITE A PROGRAM DISPLAY EVERY NTH CHARACTER FROM A FILE */

#include<stdio.h>
main()
{
            FILE *fp;
            long n;
            int count=0,i;
            clrscr();
            fp=fopen("den.txt","r");
            if(fp==NULL)
            {
              printf("\nFile Opening Error");
              exit();
            }
            while(fgetc(fp)!=EOF)
            count=count+1;
            printf("\n\nNo of Characters:%d",count);
            rewind(fp);
            printf("\nEnter Which Character U Want to print:");
            scanf("%ld",&n);
            for(i=0;i<count;i+=n)
            {
              fseek(fp,n-1L,1);
              printf("    %c",fgetc(fp));
            }
            fclose(fp);
}
```

```
/* WRITE A PROGRAM TO REPLACE EVERY NTH CHARACTER BY A SPECIFIED
CHARACTER */
#include<stdio.h>
#include<conio.h>
main( )
{
            FILE *fp;
            long n;
            int count=0,i;
            char ch,item;
            clrscr();
            fp=fopen("den.txt","r+");
            if(fp==NULL)
            {
              printf("\nFile Opening Error");
              exit();
            }
            while(fgetc(fp)!=EOF)
            count=count+1;
            rewind(fp);
            printf("\nBefore Replacing File Contents Are:\n");
            while((ch=fgetc(fp))!=EOF)
            printf("%c",ch);
            printf("\n\nNo of Characters:%d",count);
            rewind(fp);
            printf("\nEnter Which Character U Want to replace:");
            scanf("%ld",&n);
            printf("\nEvery a Character to replace:");
            scanf(" %c",&item);
            for(i=0;i<count;i+=n)
            {
              fseek(fp,n-1L,1);
              fputc(item,fp);
            }
            rewind(fp);
```

```
                printf("\n\nAfter Replacing File Contents Are:\n");
                while((ch=fgetc(fp))!=EOF)
                printf("%c",ch);
                printf("\n\nNo of Characters:%d",count);
                fclose(fp);
}
```

/* WRITE A PROGRAM TO PRINT THE CONTENTS OF A FILE IN REVERSE ORDER */

```
#include<stdio.h>
main()
{
                FILE *fp;
                long n,i;
                char ch;
                clrscr();
                fp=fopen("den.txt","r");
                if(fp==NULL)
                {
                   printf("\nFile Opening Error");
                   exit();
                }
                fseek(fp,-1L,2);
                n=ftell(fp);
                printf("\nFile Contents in Reverse Order:\n");
                for(i=1;i<=n+1;i++)
                {
                   fseek(fp,-i,2);
                   putchar(fgetc(fp));
                }
                fclose(fp);
}
```

COMMAND LINE ARGUMENTS

In environments that support C, there is a way to pass command-line arguments or parameters to a program when it begins executing. When main is called, it is called with two arguments. The first (conventionally called **argc**, for argument count) is the number of command-line arguments the program was invoked with; the second (**argv**, for argument vector) is a pointer to an array of character strings that contain the arguments, one per string. We customarily use multiple levels of pointers to manipulate these character strings.

Note that **argv[0]* is the name of the program invoked, which means that **argv[1]* is a pointer to the first argument supplied, and **argv[n]* is the last argument. If no arguments are supplied, *argc* will be one. Thus for n arguments, *argc* will be equal to n + 1. The program is called by the command line.

```
#include<stdio.h>
#include<conio.h>
main(int argc, char * argv[])
{
                int i;
                clrscr();
                printf("Number of arguments: %d\n",argc);
                printf("Name of the program :%s\n",argv[0]);
                for(i=1;i<argc;i++)
                printf("user value no %d is : %s\n",i,argv[i]);
}
```

The program is to be run at command line. The above program is executed using following steps:

a) compile the program
b) makes its exe file
c) switch to the command prompt(C:\TC>)
d) make sure that the exe file is available in the current directory.
e) type the following

```
C:\TC\cmd knreddy c faculty
Number of arguments: 4
Name of the program :C:\TC\CMD.EXE
user value no 1 is : knreddy
user value no 2 is : c
user value no 3 is : faculty
```

```
/* WRITE A PROGRAM TO PERFORM ADDITION OF GIVEN NUMBERS USING
COMMAND LINE ARGUMENTS */
#include<string.h>
main(int argc,char *argv[])
{
          int x,y;
          clrscr();
          x=atoi(argv[1]);
          y=atoi(argv[2]);
          printf("\nNumber of arguments:%d",argc);
          printf("\nAddition Result:%d",x+y);
}
```

APPLICATION OF COMMAND LINE ARGUMENTS

Using command line arguments we can create our own DOS commands like **TYPE, COPY, DEL, and RENAME.**

Let us create our own type command (which displays the contents of file on the screen)

```
#include<stdio.h>
#include<conio.h>
main(int argc, char *argv[ ])
{
    FILE *fp;
    char c;
    clrscr( );
    fp=fopen(argv[1],"r");
    if(fp==NULL)
    {
      printf("Cannot open a file");
      goto end;
    }
    printf("\n read the contents\n");
    while(!feof(fp))
    printf("%c",getc(fp));
    end:
    getch( );
}
```

Explanation:

a) Compile the above program to create an exe file
b) switch to command prompt where the exe file is present
c) type filename.exe and the file you want to read

In this example the file file_cmd.exe is located at **F:file_cmd.exe** and trying to open knr.txt file

OUTPUT:

```
F:\>file_cmd.exe knr.txt
 read the contents
this is read from command line argumen
```

```
/*copy command*/
#include<stdio.h>
#include<conio.h>
void main(int argc,char *argv[ ])
{
                FILE *fs,*fd;
                char ch;
                clrscr( )
                if(argc!=3)
                {
                   puts("Invalid number of arguments.");
                   goto end;
                }
                fs = fopen(argv[1],"r");/*pointer to source file*/
                if(fs==NULL)
                {
                   puts("Source file cannot be opened.");
                   goto end;
                }
                fd=fopen(argv[2],"w");/*pointer to destination file*/
                if (fd==NULL)
                {
                   puts("Target file cannot be opened.");
```

```
                    fclose(fs);
                    goto end;
                }
                while(1)
                {
                ch=fgetc(fs);
                if(ch==EOF)
                break;
                fputc(ch,fd);
                }
                fclose(fs);
                fclose(fd);
                printf("\ncontents of source file\n");
                fs=fopen(argv[1],"r");
                while(!feof(fs))
                printf("%c",getc(fs));
                printf("\ncontents of destination file\n");
                fd=fopen(argv[2],"r");
                while(!feof(fd))
                printf("%c",getc(fd));
                fclose(fs);
                fclose(fd);
                end:
                getch( );
}
```

OUTPUT:

```
F:\>file_copy.exe s.txt d.txt
contents of source file
c programming is very easy to learn
contents of destination file
c programming is very easy to learn
```

```
/* Write a program to implement del command with command line arguments.  Save the
program as cut.c */
#include<stdio.h>
#include<conio.h>
int main(int argc,char *argv[])
{
            FILE *f1;
            char ch;
            clrscr();
            if(argc<2)
            {
              printf("\nInvalid Arguments");
              exit( );
            }
            f1=fopen(argv[1],"r");
            if(f1==NULL)
            {
              printf("\nFile Not Available");
              exit( );
            }
            unlink(argv[1]);
            printf("\nFile has been deleted");
            return 0;
}
```

```
/* WRITE A PROGRAM TO REVERSE THE FIRST N CHARACTERS IN A FILE BY
SPECIFYING FILE NAME AND N ARE AT COMMAND LINE */
#include<stdio.h>
#include<conio.h>
main(int argc,char *argv[])
{
            FILE *f;
            char ch,x[10];
            int n,i;
            clrscr();
            f=fopen(argv[1],"r+");
            n=atoi(argv[2]);
            for(i=0;i<n;i++)
            x[i]=fgetc(f);
            rewind(f);
            for(i=n-1;i>=0;i--)
            fputc(x[i],f);
            rewind(f);
            printf("\nFile Contents Are:");
            while((ch=fgetc(f))!=EOF)
            putchar(ch);
            fclose(f);
}
```

```
/* WRITE A PROGRAM TO REVERSE THE FIRST N CHARACTERS IN A FILE BY
SPECIFYING FILE NAME AND N ARE AT COMMAND LINE */
#include<stdio.h>
#include<conio.h>
main(int argc,char *argv[ ])
{
            FILE *f;
            char ch,x[10];
            int n,i;
            clrscr();
            f=fopen(argv[1],"r+");
            n=atoi(argv[2]);
            for(i=0;i<n;i++)
            x[i]=fgetc(f);
            rewind(f);
            for(i=n-1;i>=0;i--)
            fputc(x[i],f);
            rewind(f);
            printf("\nFile Contents Are:");
            while((ch=fgetc(f))!=EOF)
            putchar(ch);
            fclose(f);
}
```