# Lecture Notes-Module Wise

## Programming for Problem Solving (KCS-101/KCS-201)

Sarvesh Kumar Swarnakar

# INDEX

## Module – 1: (Introduction to Programming)

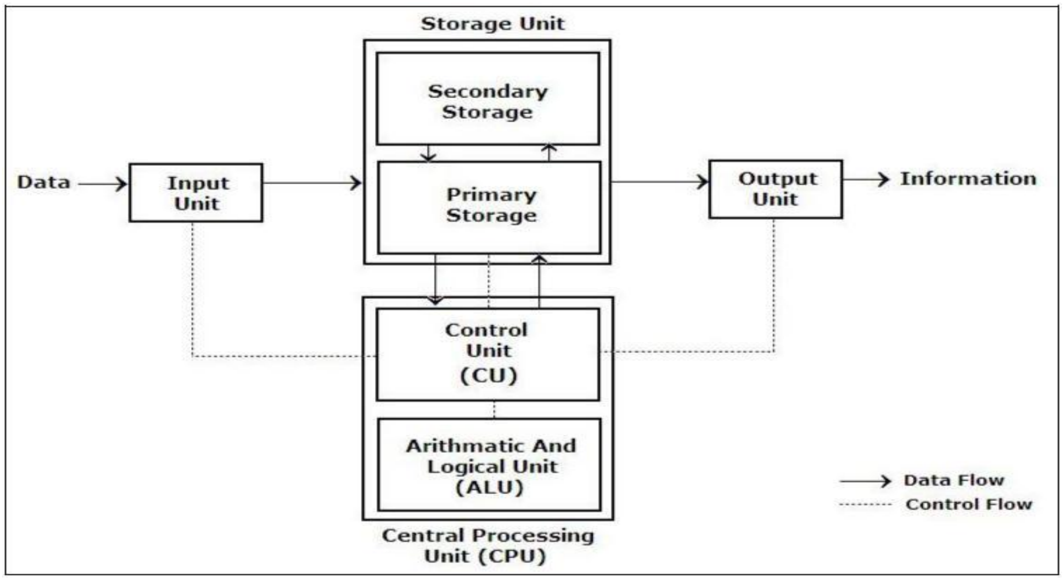
**Topic: Introduction to components of a computer system: Memory, processor, I/O Devices, storage**

### INTRODUCTION TO COMPUTERS

Any programming language is implemented on a computer. Right from its inception, to the present day, all computer system (irrespective of their shape & size) perform the following 5 basic operations. It converts the raw input data into information, which is useful to the users.

1. Input: It is the process of entering data & instructions to the computer system.
2. Storing: The data & instructions are stored for either initial or additional processing, as & when required.
3. Processing: It requires performing arithmetic or logical operation on the saved data to convert it into useful information.
4. Output: It is the process of producing the output data to the end user.
5. Controlling: The above operations have to be directed in a particular sequence to be completed.

Based on these 5 operations, we can sketch the block diagram of a computer.



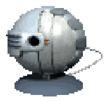
Block Diagram of a Computer

**Input Unit:**

We need to first enter the data & instruction in the computer system, before any computation begins. This task is accomplished by the input devices. (The data accepted is in a human readable form. The input device converts it into a computer readable form.

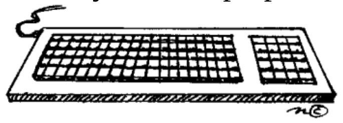
**Input device: T**he input device is the means through which data and instructions enter in a computer.

1. **Camera -** most cameras like this are used during live conversations. The camera transmits a picture from one computer to another, or can be used to record a short video.

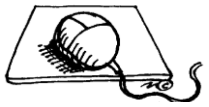


2. **Compact Disc (CD)** - CDs store information. The CD can then be put into another computer, and the information can be opened and added or used on the second computer. Note: A CD-R or CD-RW can also be used as an OUTPUT device.

3. **Keyboard -** The keyboard is a way to input letters or numbers into different applications or programs. A keyboard also has special keys that help operate the computer.

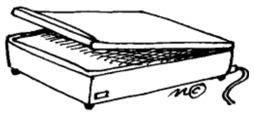


4. **Mouse -** The mouse is used to open and close files, navigate web sites, and click on a lot of commands (to tell the computer what to do) when using different applications.



5. **Microphone -** A microphone is used to record sound. The sound is then saved as a sound file on the computer.

6. **Scanner -** A scanner is used to copy pictures or other things and save them as files on the computer.



7. **Joystick -** A joystick is used to move the cursor from place to place, and to click on various items in programs. A joystick is used mostly for computer games.

8. **Bar Code Scanner** - A bar code scanner scans a little label that has a bar code on it. The information is then saved on the computer. Bar code scanners are used in libraries a lot.

**Storage Unit**:

The data & instruction that are entered have to be stored in the computer. Similarly, the end results & the intermediate results also have to be stored somewhere before being passed to the output unit. The storage unit provides solution to all these issues. This storage unit is designed to save the initial data, the intermediate result & the final result. This storage unit has 2 units: Primary storage & Secondary storage.

**Primary Storage:**

The primary storage, also called as the main memory, holds the data when the computer is currently on. As soon as the system is switched off or restarted, the information held in primary storage disappears (i.e. it is volatile in nature). Moreover, the primary storage normally has a limited storage capacity, because it is very expensive as it is made up of semiconductor devices.

**Secondary Storage:**

The secondary storage, also called as the auxiliary storage, handles the storage limitation & the volatile nature of the primary memory. It can retain information even when the system is off. It is basically used for holding the program instructions & data on which the computer is not working on currently, but needs to process them later.

**Magnetic Disk**

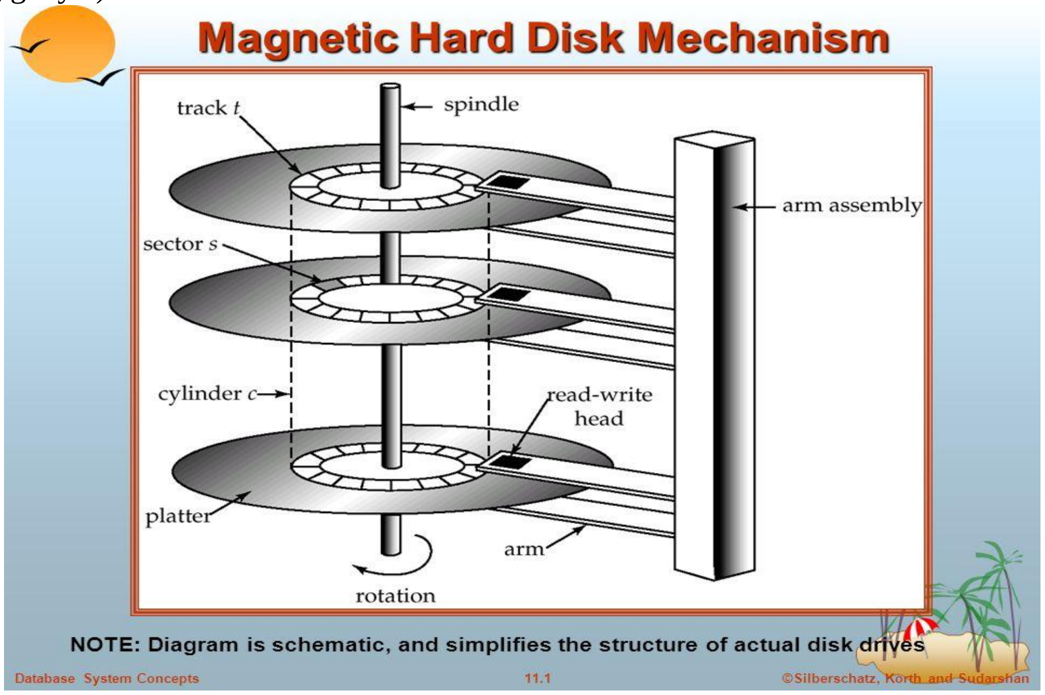
The Magnetic Disk is Flat, circular platter with metallic coating that is rotated beneath read/write heads. It is a Random access device; read/write head can be moved to any location on the platter.

**Floppy Disk**

These are small removable disks that are plastic coated with magnetic recording material. Floppy disks are typically 3.5″ in size (diameter) and can hold 1.44 MB of data. This portable storage device is a rewritable media and can be reused a number of times. Floppy disks are commonly used to move files between different computers. The main disadvantage of floppy disks is that they can be damaged easily and, therefore, are not very reliable.

**Hard disk.**

A hard disk consists of one or more rigid metal plates coated with a metal oxide material that allows data to be magnetically recorded on the surface of the platters. The hard disk platters spin at a high rate of speed, typically 5400 to 7200 revolutions per minute (RPM).Storage capacities of hard disks for personal computers range from 10 GB to 120 GB (one billion bytes are called a gigabyte).


Magnetic Hard Disk Mechanism
track t
spindle
arm assembly
sector s
cylinder c
read-write head
platter
arm
rotation
NOTE: Diagram is schematic, and simplifies the structure of actual disk drives
Database System Concepts
11.1
©Silberschatz, Korth and Sudarshan

**CD:**

Compact Disk (CD) is portable disk having data storage capacity between 650-700 MB. It can hold large amount of information such as music, full-motion videos, and text etc. It contains digital information that can be read, but cannot be rewritten. Separate drives exist for reading and writing

CDs. Since it is a very reliable storage media, it is very often used as a medium for distributing large amount of information to large number of users. In fact today most of the software is distributed through CDs.

**DVD**

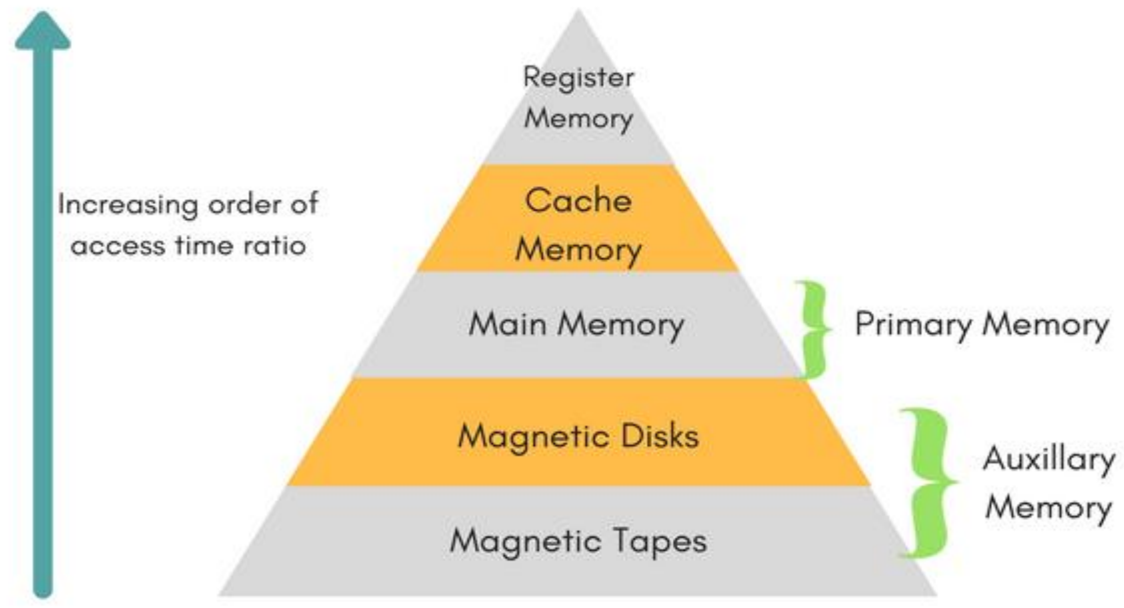
Digital Versatile Disk (DVD) is similar to a CD but has larger storage capacity and enormous clarity. Depending upon the disk type it can store several Gigabytes of data (as opposed to around 650MB of a CD). DVDs are primarily used to store music or movies and can be played back on your television or the computer too. They are not rewritable media. It’s also termed DVD (Digital Video Disk)

**Memory Organization in Computer**

A memory unit is the collection of storage units or devices together. The memory unit stores the binary information in the form of bits. Generally, memory/storage is classified into 2 categories:

- **Volatile Memory**: This loses its data, when power is switched off.
- **Non-Volatile Memory**: This is a permanent storage and does not lose any data when power is switched off.

**Memory Hierarchy**



The total memory capacity of a computer can be visualized by hierarchy of components. The memory hierarchy system consists of all storage devices contained in a computer system from the slow Auxiliary Memory to fast Main Memory and to smaller Cache memory.

**Auxillary memory** access time is generally **1000 times** that of the main memory, hence it is at the bottom of the hierarchy.

The **main memory** occupies the central position because it is equipped to communicate directly with the CPU and with auxiliary memory devices through Input/output processor (I/O).

When the program not residing in main memory is needed by the CPU, they are brought in from auxiliary memory. Programs not currently needed in main memory are transferred into auxiliary memory to provide space in main memory for other programs that are currently in use.

The **cache memory** is used to store program data which is currently being executed in the CPU. Approximate access time ratio between cache memory and main memory is about **1 to 7~10**

**Central Processing Unit:**

Together the Control Unit & the Arithmetic Logic Unit are called as the Central Processing Unit (CPU). The CPU is the brain of the computer. Like in humans, the major decisions are taken by the brain itself & other body parts function as directed by the brain. Similarly in a computer system, all the major calculations & comparisons are made inside the CPU. The CPU is responsible for activating & controlling the operation of other units of the computer system.

**Arithmetic Logic Unit:**

The actual execution of the instructions (arithmetic or logical operations) takes place over here. The data & instructions stored in the primary storage are transferred as & when required. No processing is done in the primary storage. Intermediate results that are generated in ALU are temporarily transferred back to the primary storage, until needed later. Hence, data may move from the primary storage to ALU & back again to storage, many times, before the processing is done.

**Control Unit:**

This unit controls the operations of all parts of the computer but does not carry out any actual data processing.It is responsible for the transfer of data and instructions among other units of the computer.It manages and coordinates all the units of the system.It also communicates with Input/Output devices for transfer of data or results from the storage units.

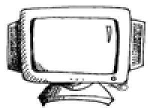
**OutputUnit:**

The job of an output unit is just the opposite of an input unit. It accepts the results produced by the computer in coded form. It converts these coded results to human readable form. Finally, it displays the converted results to the outside world with the help of output devices ( Eg :monitors, printers, projectors etc..).

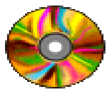
**Output device:**

Device that lets you see what the computer has accomplished.

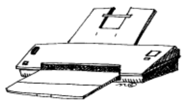
1. **Monitor -** A monitor is the screen on which words, numbers, and graphics can be seem. The monitor is the most common output device.



2. **Compact Disk** - Some compact disks can be used to put information on. This is called burning information to a CD. NOTE: A CD can also be an input device.



3. **Printer -** A printer prints whatever is on the monitor onto paper. Printers can print words, numbers, or pictures.



4. **Speaker -** A speaker gives you sound output from your computer. Some speakers are built into the computer and some are separate.

5. **Headphones -** Headphones give sound output from the computer. They are similar to speakers, except they are worn on the ears so only one person can hear the output at a time.

**Hardware-**
This hardware is responsible for all the physical work of the computer.
**Software-**
This software commands the hardware what to do & how to do it. Together, the hardware & software form the computer system. This software is further classified as system software & application software.

**System Software-**
System software are a set of programs, responsible for running the computer, controlling various operations of computer systems and management of computer resources. They act as an interface between the hardware of the computer & the application software. E.g.: Operating System.
**Application software**
is a set of programs designed to solve a particular problem for users. It allows the end user to do something besides simply running the hardware. E.g.: Web Browser, Gaming Software, etc.

**Computer Classification:**
Computers can be generally classified by size and power as follows, though there is considerable overlap:

**a. Personal computer:** a small, single-user computer based on a microprocessor. In addition to the microprocessor, a personal computer has a keyboard for entering data, a monitor for displaying information, and a storage device for saving data.

**b. Workstation:** a powerful, single-user computer. A workstation is like a personal computer, but it has a more powerful microprocessor and a higher-quality monitor.

**c. Minicomputer:** a multi-user computer capable of supporting from 10 to hundreds of users simultaneously.

**d. Mainframe:** a powerful multi-user computer capable of supporting many hundreds or thousands of users simultaneously.

**e. Supercomputer:** an extremely fast computer that can perform hundreds of millions of instructions per second.

**For Reference Only **Digital Computer-A brief History**

Each generation of computer is characterized by a major technological development that fundamentally changed the way computers operate, resulting in increasingly smaller, cheaper, more powerful and more efficient and reliable devices.

**First Generation (1940-1956) Vacuum Tubes**

- The first computers used vacuum tubes for circuitry and magnetic drums for memory, and were often enormous, taking up entire rooms.
- They were very expensive to operate and in addition to using a great deal of electricity, generated a lot of heat, which was often the cause of malfunctions.
- First generation computers relied on machine language, the lowest-level programming language understood by computers, to perform operations, and they could only solve one problem at a time.
- Input was based on punched cards and paper tape, and output was displayed on printouts.
- The UNIVAC and ENIAC computers are examples of first-generation computing devices.

**Second Generation (1956-1963) Transistors**

- Transistors replaced vacuum tubes and ushered in the second generation of computers. The transistor was far superior to the vacuum tube, allowing computers to become smaller, faster, cheaper, more energy-efficient and more reliable than their first-generation predecessors.
- Second-generation computers still relied on punched cards for input and printouts for output.
- Second-generation computers moved from cryptic binary machine language to symbolic, or assembly, languages, which allowed programmers to specify instructions in words. High-level programming languages were also being developed at this time, such as early versions of COBOL and FORTRAN.
- The first computers of this generation were developed for the atomic energy industry.

**Third Generation (1964-1971) Integrated Circuits**

- The development of the integrated circuit was the hallmark of the third generation of computers.
- Transistors were miniaturized and placed on silicon chips, called semiconductors, which drastically increased the speed and efficiency of computers.
- Instead of punched cards and printouts, users interacted with third generation computers through keyboards and monitors and interfaced with an operating system, which allowed the device to run many different applications at one time with a central program that monitored the memory.
- Computers for the first time became accessible to a mass audience because they were smaller and cheaper than their predecessors.

**Fourth Generation (1971-Present) Microprocessors**

- The microprocessor brought the fourth generation of computers, as thousands of integrated circuits were built onto a single silicon chip.
- The Intel 4004 chip, developed in 1971, located all the components of the computer—from the central processing unit and memory to input/output controls—on a single chip.
- As these small computers became more powerful, they could be linked together to form networks, which eventually led to the development of the Internet.
- Fourth generation computers also saw the development of GUIs, the mouse and handheld devices.

**Fifth Generation (Present and Beyond) Artificial Intelligence**

- Fifth generation computing devices, based on artificial intelligence, are still in development, though there are some applications, such as voice recognition, that are being used today.
- The use of parallel processing and superconductors is helping to make artificial intelligence a reality.

## Topic: Operating system –functions and services

**Notes**

**Definition:** An operating system (OS) is a collection of software that manages computer hardware resources and provides common services for computer programs.

**Definition:**(1)As an interface (2) As an environment (3) As a system software

**Operating System functions:**

|  |  |  |
|---|---|---|
| **1.** Process Management | **2.**Main Memory management | **3.**I/O device management |
| **4.** File Management | **5.**Secondary storage management | **6.**Network Management |
| **7.** System Protection | **8.**Command interpretation |  |

**Classification of operating systems:**

**(i)Batch System:**

**(a)** Program, its related data and relevant control command should be submitted together, in the form of a job.
**(b)** No interaction between the users and the executing programs, very simple, transfer control automatically.
**(c)** Scheduling of jobs is in the order of FCFS
**(d)**Memory management is done by Master/Slave concept.
**(e)**No need of concurrency control.
**(f)** Good for the programs which have long execution time.

**(ii)Interactive System:**

An operating system that allows users to run interactive programs. **Pretty much all operating systems that are on PCs are interactive OS.**

**(iii)Time Sharing System:**

**1.** Time sharing (multitasking) is a **logical extension** of multiprogramming. The CPU executes multiple jobs by switching among them, but the switches occur so frequently that the users can interact with each program while it is running
**2**. CPU bound is divided into different time slots depending upon the number of users using the system.
**3.** There must be some CPU scheduling, memory management scheme, synchronization and communication schemes.

**(iv)Real Time System:**

**1.** This type of operating systems are used to control Scientific devices and similar small instruments where memory and resources are crucial. These type of devices have very limited or no end user utilities , so more effort should go into making the OS really memory efficient and fast (less coding), so as to minimize the execution time ,in turn saving on power as well. E x: VHDL, 8086 etc.
**2.** Provide quick response time and thus to meet a scheduling deadline (time constraint).**3.** Resource utilization is secondary concern to these systems.
**4.** Applicable to Rocket launching, flight control, robotics.**5.** Types (a) soft RTS (b) hard RTS

**(v)Multiprocessor system:**

**1.** Multiprocessor System consists of a set of processors that shares a set of physical memory blocks over an interconnection network.
**2.** Controls and manage the hardware and software resources such that user can view the entire system as a powerful uniprocessor system.

**3.** Design is too complex.
**4.** Parallel or tightly coupled systems.

**(vi)Multiuser System:**
**1. Multi-user** is a term that defines an operating system or application software that allows concurrent access by multiple users of a computer.
**2.** Time-sharing systems are multi-user systems. Most batch processing systems for mainframe computers may also be considered "multi-user", to avoid leaving the CPU idle while it waits for I/O operations to complete.

**Common services provided by an operating system** −

- Program execution
- I/O operations
- File System manipulation
- Communication
- Error Detection
- Resource Allocation
- Protection

****Additional Informations about OS-**

**DOS (Disk Operating System)**

1. Microsoft Disk operating system, MS-DOS is a non-graphical command line operating system derived from 86-DOS that was created for IBM compatible computers. MS-DOS originally written by Tim Paterson and introduced by Microsoft in August 1981 and was last updated in 1994 .
2. Today, MS-DOS is no longer used; however, the command shell, more commonly known as the Windows command line is still used by many users. In the picture to the right, is an example of what a MS-DOS window more appropriately referred to as the Windows command line looks like under Microsoft Windows.
3. DOS (an acronym for Disk Operation System) is a tool which allows you to control the operation of the IBM PC. DOS is software which was written to control hardware.

**Essential DOS Commands and Concepts**

**1. Change the Default Drive**

To change the default drive, simply type the letter of the choice. The new default will be listed in subsequent DOS prompts.

**2. CHDIR (CD) Change Directory Command**

Once you have located the directory you want, you may move from directory to directory using the CD command (change directory)

**3. COPY Command**

The COPY command can be used both to copy files from disk to disk or to create a second copy of a file on a single disk. (There are many more uses of the COPY command, but only the basic operation is discussed here.)

**4. DIR (Directory) Command**

The DIRECTORY command lists the names and sizes of all files located on a particular disk.

**5. DIR Options**

Two little characters, '*' and '?', will make your life with computers much easier. Their use is illustrated below.

**6. ERASE Command**

The ERASE command deletes specified files.

7. **FORMAT Command**

You must format new disks before using them on the IBM computers. The format command checks a diskette for flaws and creates a directory where all the names of the diskette's files will be stored.

8. **MKDIR (MD) Make Directory Command**

This command creates a new directory.

9. **RENAME (REN) Command**

The RENAME command permits users to change the name of a file without making a copy of it.

10. **RMDIR (RD) Remove Directory Command**

This command removes a directory. It is only possible to execute this command if the directory you wish to remove is empty.

11. **Stop Execution (Ctrl-Break)**

If you wish to stop the computer in the midst of executing the current command, you may use the key sequence Ctrl-Break. Some other commands are: DATE, TIME, VER, CLS, and COPYCON.

**Microsoft Windows**

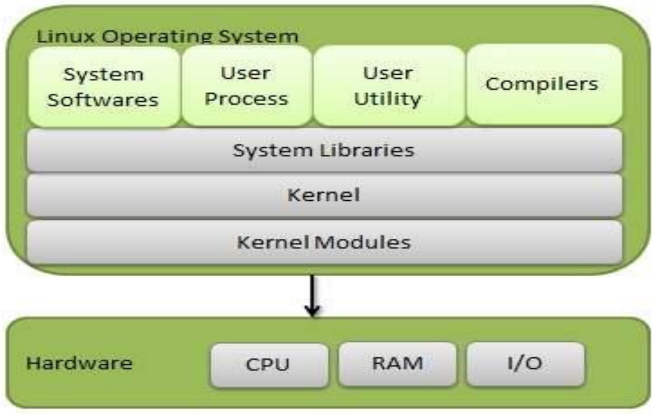
- Windows is a meta family of graphical operating systems developed, marketed, and sold by Microsoft.
- Active Windows families include Windows NT, Windows Embedded and Windows Phone; these may encompass subfamilies, e.g. Windows Embedded Compact (Windows CE) or Windows Server. Defunct Windows families include Windows 9x; Windows 10 Mobile is an active product, unrelated to the defunct family Windows Mobile.
- Microsoft introduced an operating environment named Windows on November 20, 1985, as a graphical operating system shell for MS-DOS in response to the growing interest in graphical user interfaces (GUIs).
- Version history-Windows 1.0, Windows 2.0, and Windows 2.1x➔Windows 3.x➔**Windows 9x**➔Windows NT➔ Windows XP➔Windows Vista➔**Windows 7**➔**Windows 8 and 8.1**➔**Windows 10**➔

**Basic Features** of Windows Operating System-

1) Windows Easy Transfer 2) Windows Anytime Upgrade3) Windows Basics 4) Searching and Organizing 5) Parental Controls 6) Ease of Access Center 7) Default Programs 8) Remote Desktop Connection

**Linux**

- The Unix operating system was conceived and implemented in 1969 at AT&T's Bell Laboratories in the United States by Ken Thompson, Dennis Ritchie, Douglas McIlroy, and Joe Ossanna.
- Linux Operating System has primarily three components
    - **Kernel** − Kernel is the core part of Linux. It is responsible for all major activities of this operating system. It consists of various modules and it interacts directly with the underlying hardware. Kernel provides the required abstraction to hide low level hardware details to system or application programs.
    - **System Library** − System libraries are special functions or programs using which application programs or system utilities accesses Kernel's features. These libraries implement most of the functionalities of the operating system and do not requires kernel module's code access rights.
    - **System Utility** − System Utility programs are responsible to do specialized, individual level tasks.



**Basic Features**

Following are some of the important features of Linux Operating System.

- **Portable** − Portability means software can works on different types of hardware in same way. Linux kernel and application programs supports their installation on any kind of hardware platform.
- **Open Source** − Linux source code is freely available and it is community based development project. Multiple teams work in collaboration to enhance the capability of Linux operating system and it is continuously evolving.
- **Multi-User** − Linux is a multiuser system means multiple users can access system resources like memory/ ram/ application programs at same time.
- **Multiprogramming** − Linux is a multiprogramming system means multiple applications can run at same time.
- **Hierarchical File System** − Linux provides a standard file structure in which system files/ user files are arranged.
- **Shell** − Linux provides a special interpreter program which can be used to execute commands of the operating system. It can be used to do various types of operations, call application programs. etc.
- **Security** − Linux provides user security using authentication features like password protection/ controlled access to specific files/ encryption of data.

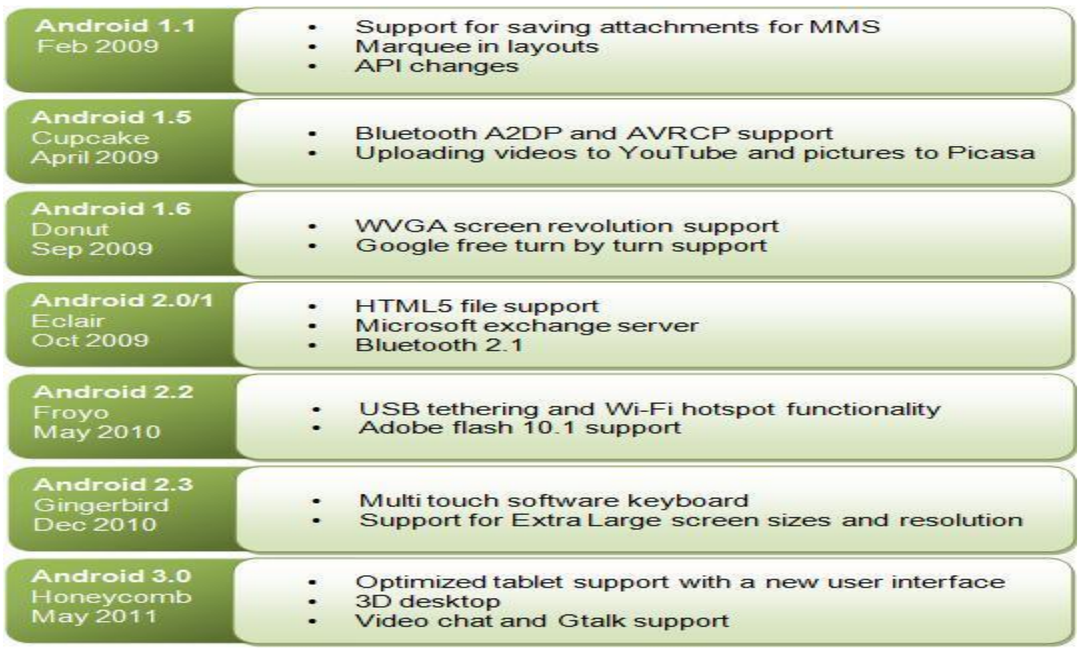
**Architecture**

The architecture of a Linux System consists of the following layers −

- **Hardware layer** − Hardware consists of all peripheral devices (RAM/ HDD/ CPU etc).
- **Kernel** − It is the core component of Operating System, interacts directly with hardware, provides low level services to upper layer components.
- **Shell** − An interface to kernel, hiding complexity of kernel's functions from users. The shell takes commands from the user and executes kernel's functions.
- **Utilities** − Utility programs that provide the user most of the functionalities of an operating systems.

**Android**

**What is Android-**One of the most widely used mobile OS these days is **ANDROID**. **Android** is a software bunch comprising not only operating system but also middleware and key applications. Android Inc was founded in Palo Alto of California, U.S. by Andy Rubin, Rich miner, Nick sears and Chris White in 2003. Later Android Inc. was acquired by Google in 2005. After original release there have been number of updates in the original version of Android



| | |
|---|---|
| Android 1.1<br>Feb 2009 | • Support for saving attachments for MMS<br>• Marquee in layouts<br>• API changes |
| Android 1.5<br>Cupcake<br>April 2009 | • Bluetooth A2DP and AVRCP support<br>• Uploading videos to YouTube and pictures to Picasa |
| Android 1.6<br>Donut<br>Sep 2009 | • WVGA screen revolution support<br>• Google free turn by turn support |
| Android 2.0/1<br>Eclair<br>Oct 2009 | • HTML5 file support<br>• Microsoft exchange server<br>• Bluetooth 2.1 |
| Android 2.2<br>Froyo<br>May 2010 | • USB tethering and Wi-Fi hotspot functionality<br>• Adobe flash 10.1 support |
| Android 2.3<br>Gingerbird<br>Dec 2010 | • Multi touch software keyboard<br>• Support for Extra Large screen sizes and resolution |
| Android 3.0<br>Honeycomb<br>May 2011 | • Optimized tablet support with a new user interface<br>• 3D desktop<br>• Video chat and Gtalk support |

•

**Features & Specifications**

**Android** is a powerful Operating System supporting a large number of applications in Smart Phones. These applications make life more comfortable and advanced for the users.

Android applications are written in java programming language.

Android is available as open source for developers to develop applications which can be further used for selling in android market.

For software development, Android provides **Android SDK** (Software development kit).

**Applications**

These are the basics of Android applications:

- Android applications are composed of one or more application components (activities, services, content providers, and broadcast receivers)
- Each component performs a different role in the overall application behavior, and each one can be activated individually (even by other applications)
- The manifest file must declare all components in the application and should also declare all application requirements, such as the minimum version of Android required and any hardware configurations required
- Non-code application resources (images, strings, layout files, etc.) should include alternatives for different device configurations (such as different strings for different languages)

## Topic: Concept of assembler, compiler, interpreter, loader and linker

- A language that is acceptable to a computer system is called a **computer language** or **programming language** and the process of creating a sequence of instructions in such a language is called **programming** or **coding**.
- A program is a set of instructions, written to perform a specific task by the computer. A set of large program is called **software**. To develop software, one must have knowledge of a programming language.
- Before moving on to any programming language, it is important to know about the various types of languages used by the computer.

### COMPUTER LANGUAGES

- Languages are a means of communication. Normally people interact with each other through a language. On the same pattern, communication with computers is carried out through a language. This language is understood both by the user and the machine.
- Just as every language like English, Hindi has its own grammatical rules; every computer language is also bounded by rules known as syntax of that language. The user is bound by that syntax while communicating with the computer system.

Computer languages are broadly classified as:

- **Low Level Language**: The term low level highlights the fact that it is closer to a language which the machine understands

  The low level languages are classified as:
- **Machine Language**: This is the language (in the form of 0's and 1's, called binary numbers) understood directly by the computer. It is machine dependent. It is difficult to learn and even more difficult to write programs.
- **Assembly Language**: This is the language where the machine codes comprising of 0'sand 1's are substituted by symbolic codes (called mnemonics) to improve their understanding. It is the first step to improve programming structure. Assembly language programming is simpler and less time consuming than machine level programming, it is easier to locate and correct errors in assembly language than in machine language programs. It is also machine dependent. Programmers must have knowledge of the machine on which the program will run.

- **High Level Language:** Low level language requires extensive knowledge of the hardware since it is machine dependent. To overcome this limitation, high level language has been evolved which uses normal English, which is easy to understand to solve any problem. High level languages are computer independent and programming becomes quite easy and simple. Various high level languages are given below:
- BASIC (Beginners All Purpose Symbolic Instruction Code): It is widely used, easy to learn general purpose language. Mainly used in microcomputers in earlier days.
- COBOL (Common Business Oriented language): A standardized language used for commercial applications.
- FORTRAN (Formula Translation): Developed for solving mathematical and scientific problems. One of the most popular languages among scientific community.
- C: Structured Programming Language used for all purpose such as scientific application, commercial application, developing games etc.

- C++: Popular object oriented programming language, used for general purpose.

**PROGRAMMING LANGUAGE TRANSLATORS**

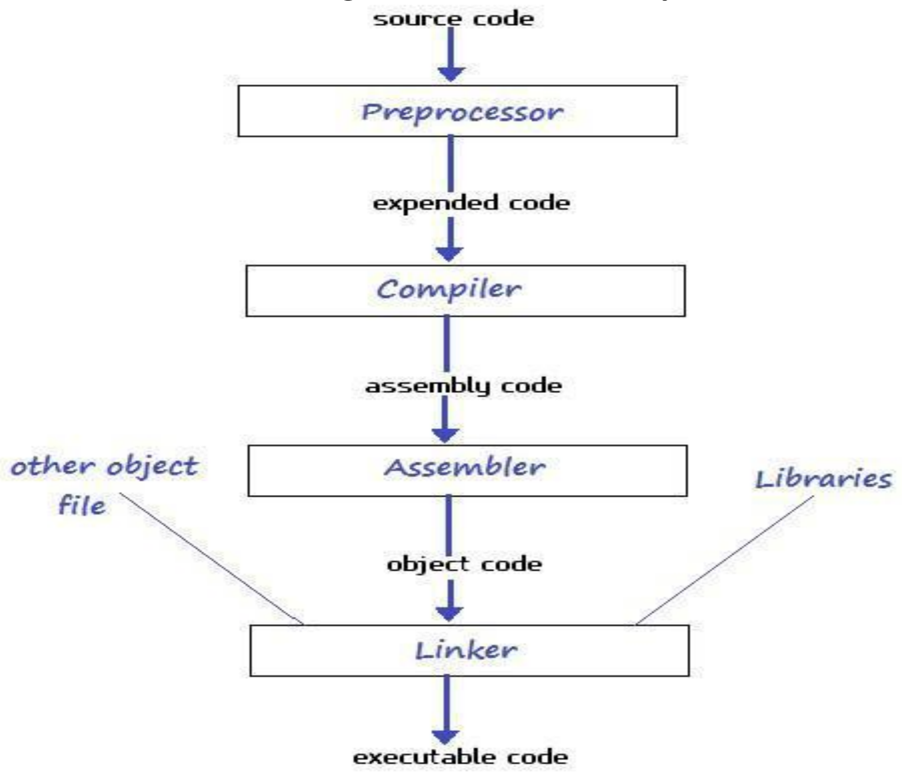
- As you know that high level language is machine independent and assembly language though it is machine dependent yet mnemonics that are being used to represent instructions are not directly understandable by the machine. Hence to make the machine understand the instructions provided by both the languages, programming language instructors are used.
- They transform the instruction prepared by programmers into a form which can be interpreted & executed by the computer. Flowing are the various tools to achieve this purpose:

**Compiler**: The software that reads a program written in high level language and translates it into an equivalent program in machine language is called as compiler. The program written by the programmer in high level language is called source program and the program generated by the compiler after translation is called as object program.

**Interpreter**: it also executes instructions written in a high level language. Both complier & interpreter have the same goal i.e. to convert high level language into binary instructions, but their method of execution is different. The complier converts the entire source code into machine level program, while the interpreter takes 1 statement, translates it, executes it & then again takes the next statement.

**Assembler**: The software that reads a program written in assembly language and translates it into an equivalent program in machine language is called as assembler.

**Linker**: A linker or link editor is a computer program that takes one or more object files generated by a compiler and combines them into a single executable file, library file, or another object file.



**Topic: Idea of Algorithm: Representation of Algorithm, Flowchart, Pseudo code with examples, From algorithms to programs, source code.**

## Algorithm

- Characteristics of algorithm: Input, output, definiteness, finiteness, effectiveness.
- Algorithm is a step-by-step solution to a problem. Example: one algorithm for adding two digit numbers is "add the units, add the tens and combine the answers"
- "Algorithm" is named after the 9th century Persian mathematician Al-Khwarizmi.

**Example: For factorial**

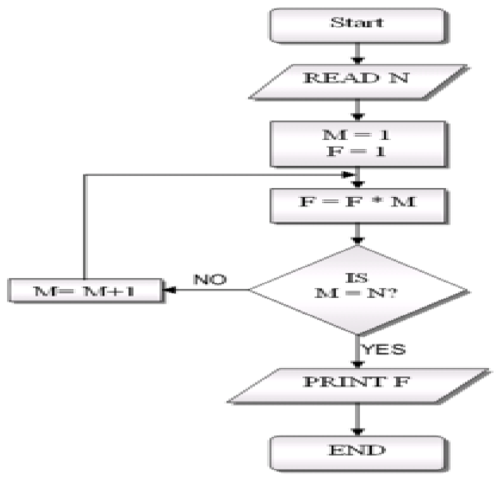
1. Start
2. Input number N (for which we want to calculate factorial.)
3. Let M=1 and F=1.
4. F=F*M.
5. Is M=N?
6. If NO then M=M+1 and go to step 3.
7. If YES then output F
8. End

## Flow Chart:

A flowchart is a type of diagram that represents an algorithm or process, showing the steps as boxes of various kinds, and their order by connecting these with arrows. This diagrammatic representation can give a step-by-step solution to a given problem.

| Symbol | Representation | | Symbol | Representation |
|---|---|---|---|---|
| ⬭ | Start/Stop | | ◇ | Decision |
| ▭ | Process | | ○ | Connector |
| ▱ | Input/Output | | ↓ | Flow Direction |

**Example: For factorial.**



**Advantages Of Using FLOWCHARTS: -**

- Communication: - Flowcharts are better way of communicating the logic of a system to all concerned.
- Effective analysis: - With the help of flowchart, problem can be analyzed in more effective way.

- Proper documentation: - Program flowcharts serve as a good program documentation, which is needed for various purposes.
- Efficient Program Maintenance: - The maintenance of operating program becomes easy with the help of flowchart. It helps the programmer to put efforts more efficiently on that part

**Disadvantages Of Using FLOWCHARTS: -**

**1.** Complex logic: - Sometimes, the program logic is quite complicated. In that case, flowchart becomes complex and clumsy.

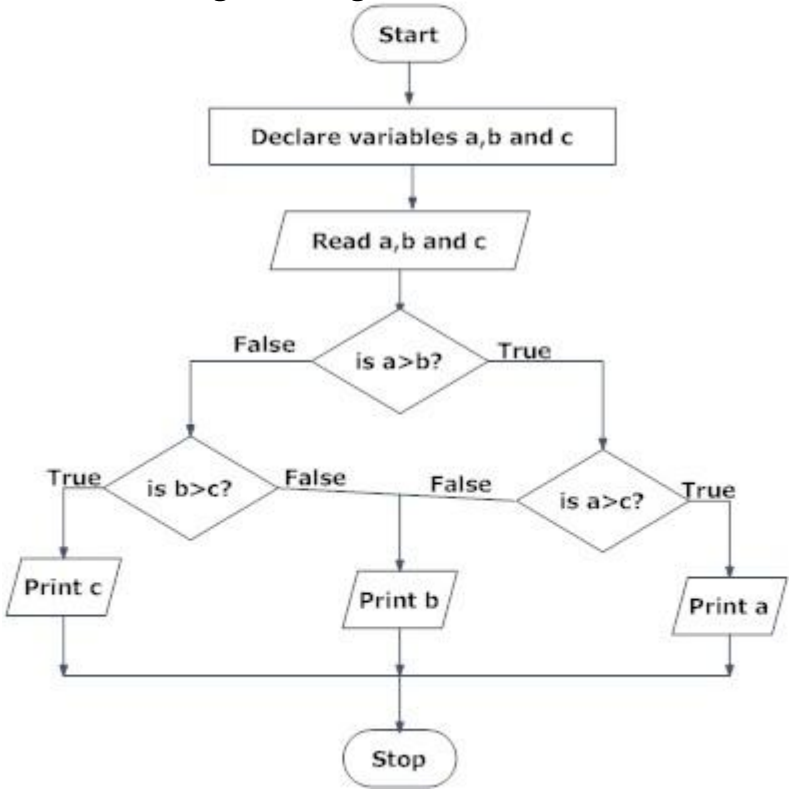
**2.** Alterations and Modifications: - If alterations are required the flowchart may require re-drawing completely.

**3.** Reproduction: - As the flowchart symbols cannot be typed, reproduction of flowchart becomes a problem.

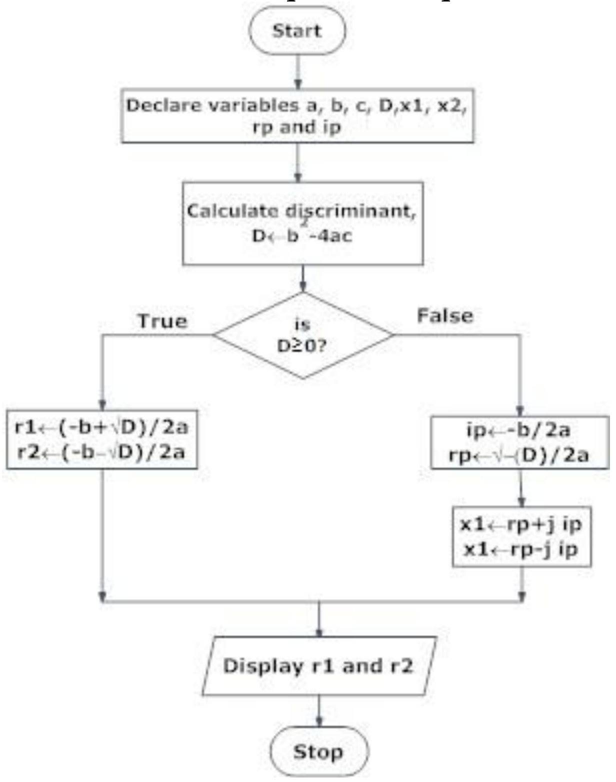
**4.** The essentials of what is done can easily be lost in the technical details of how it is done.

**Examples of Algorithm and Flowchart (Random)-For hint**

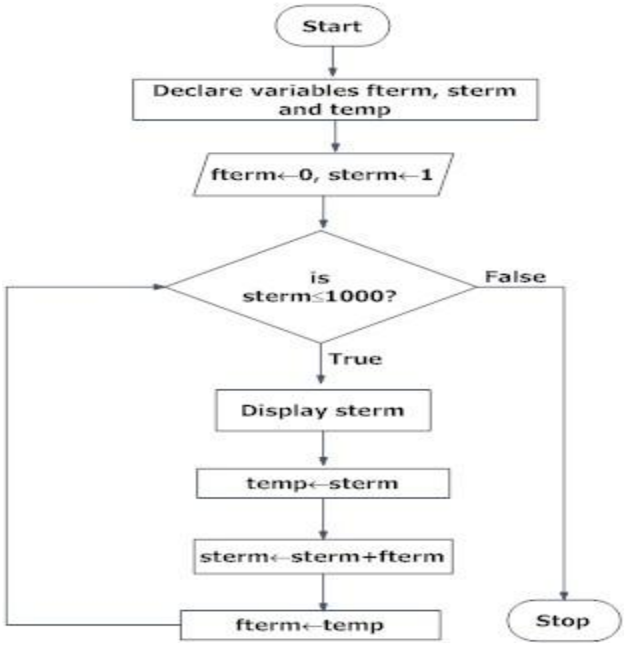
**1. Draw flowchart to find the largest among three different numbers entered by user.**



**2.Draw a flowchart to find all the roots of a quadratic equation $ax^2+bx+c=0$**



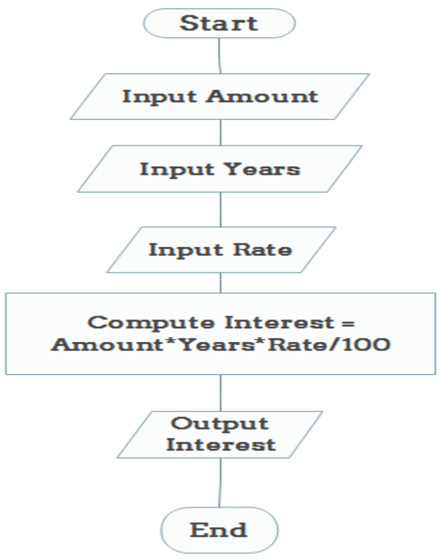
**3.Draw a flowchart to find the Fibonacci series till term≤1000.**



**Example : Calculate the Interest of a Bank Deposit**
**Algorithm:**

- Step 1: Read amount,
- Step 2: Read years,
- Step 3: Read rate,
- Step 4: Calculate the interest with formula "Interest=Amount*Years*Rate/100
- Step 5: Print interest,

**Flowchart:**

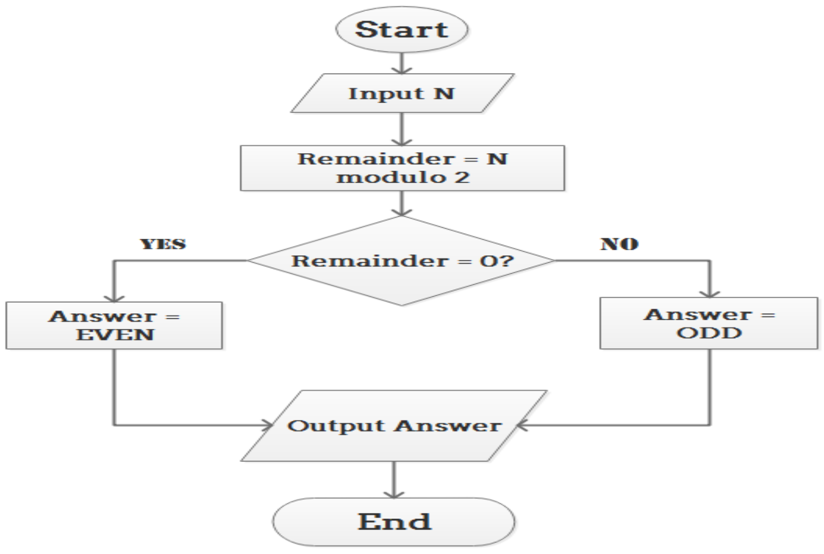


**Example : Determine and Output Whether Number N is Even or Odd**

**Algorithm:**

- Step 1: Read number N,
- Step 2: Set remainder as N modulo 2,
- Step 3: If remainder is equal to 0 then number N is even, else number N is odd,
- Step 4: Print output.

**Flowchart:**

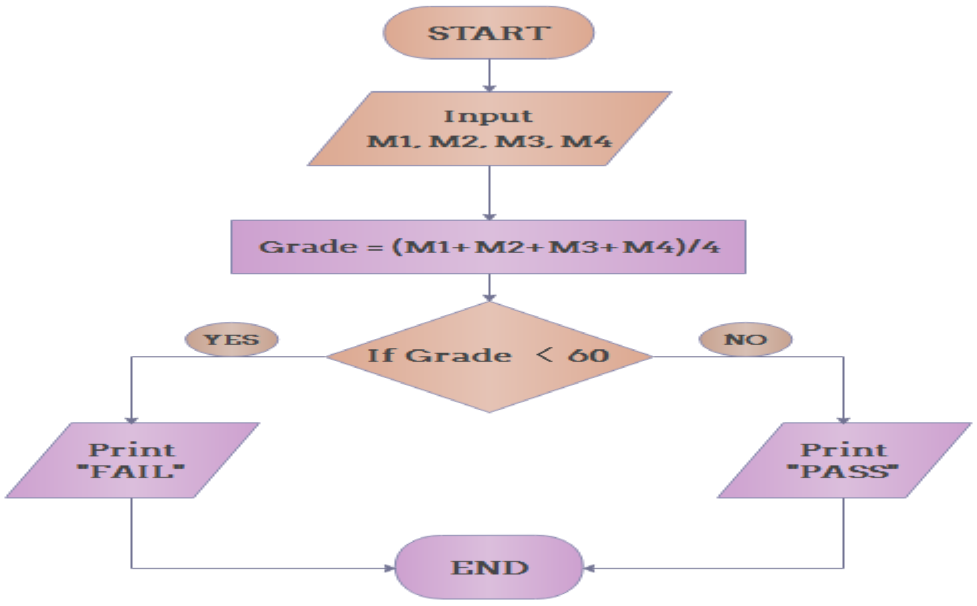


**Example : Determine Whether A Student Passed the Exam or Not:**

**Algorithm:**

- Step 1: Input grades of 4 courses M1, M2, M3 and M4,
- Step 2: Calculate the average grade with formula "Grade=(M1+M2+M3+M4)/4"
- Step 3: If the average grade is less than 60, print "FAIL", else print "PASS".

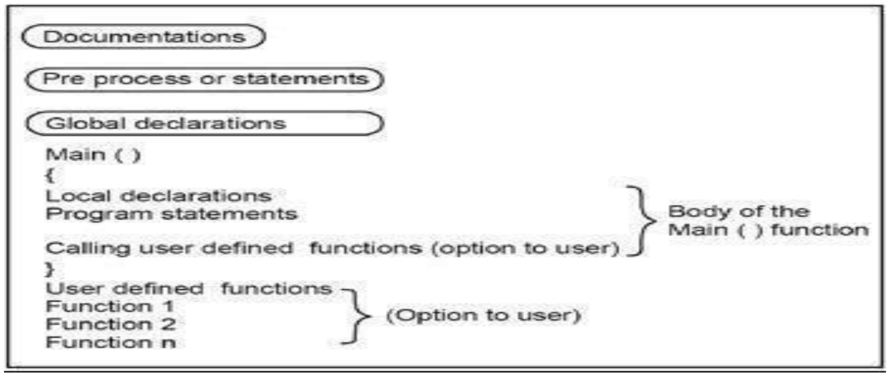
**Flowchart:**



**Topic: Programming Basics: Structure of C program, Syntax , errors**

## STRUCTURE OF A C PROGRAM

The structure of a C program is a protocol (rules) to the programmer, which he has to follow while writing a C program. The general basic structure of C program is shown in the figure below.



Based on this structure, we can sketch a C program.

Example:

```
/* This program accepts a number & displays it to the user*/
#include <stdio.h> void main(void)
{ int number;
printf( "Please enter a number: " );
scanf( "%d", &number );
printf( "You entered %d", number );
return 0;
            }
```

Stepwise explanation:

**#include**

The part of the compiler which actually gets your program from the source file is called the preprocessor.

**#include <stdio.h>**

#include is a pre-processor directive. It is not really part of our program, but instead it is an instruction to the compiler to make it do something. It tells the C compiler to include the contents of a file (in this case the system file called stdio.h).

The compiler knows it is a system file, and therefore must be looked for in a special place, by the fact that the filename is enclosed in <> characters

**<stdio.h>**

stdio.h is the name of the standard library definition file for all Standard Input and Output functions.

Your program will almost certainly want to send information to the screen and read things from the keyboard, and stdio.h is the name of the file in which the functions that we want to use are defined.

The function we want to use is called printf. The actual code of printf will be tied in later by the linker.

The ".h" portion of the filename is the language extension, which denotes an include file.

**void**

This literally means that this means nothing. In this case, it is referring to the function whose name follows.Void tells to C compiler that a given entity has no meaning, and produces no error.

**Main** In this particular example, the only function in the program is called main. A C program is typically made up of large number of functions. Each of these is given a name by the programmer and they refer to each other as the program runs.C regards the name main as a special case and will run this function first i.e. the program execution starts from main. A parameter to a function gives the function something to work on.

**{ (Brace)**

This is a brace (or curly bracket). As the name implies, braces come in packs of two - for every open brace there must be a matching close one. Braces allow us to group pieces of program together, often called a block. A block can contain the declaration of variable used within it, followed by a sequence of program statements. In this case the braces enclose the working parts of the function main.

**;( semicolon)**

The semicolon marks the end of the list of variable names, and also the end of that declaration statement.

All statements in C programs are separated by ";" (semicolon) characters. The ";" character is actually very important. It tells the compiler where a given statement ends. If the compiler does not find one of these characters where it expects to see one, then it will produce an error.

**scanf**

In other programming languages, the printing and reading functions are a part of the language. In C this is not the case; instead they are defined as standard functions which are part of the language specification, but are not a part of the language itself. The standard input/output library contains a number of functions for formatted data transfer; the two we are going to use are scanf (scan formatted) and printf (print formatted).

**printf**

The printf function is the opposite of scanf. It takes text and values from within the program and sends it out onto the screen. Just like scanf, it is common to all versions of C and just like scanf, it is described in the system file stdio.h.The first parameter to a printf is the format string, which contains text, value descriptions and formatting instructions.

## FILES USED IN A C PROGRAM

**Source File-** This file contains the source code of the program. The file extension of any c file is **.c**. The file contains C source code that defines the main function & maybe other functions.

**Header File-** A header file is a file with extension **.h** which contains the C function declarations and macro definitions and to be shared between several source files.

**Object File-** An object file is a file containing object code, with an extension **.o**, meaning relocatable format machine code that is usually not directly executable. Object files are produced by an assembler, compiler, or other language translator, and used as input to the linker, which in turn typically generates an executable or library by combining parts of object files.

**Executable File-** The binary executable file is generated by the linker. The linker links the various object files to produce a binary file that can be directly executed.

### Types of error

a) Syntax Errors: Errors in syntax (grammar) of the program.
b) Semantic Errors: Errors in the meaning of the program.
c) Logical Errors: Errors in logic of the program. Compiler cannot diagnose these kinds of errors.
d) Runtime Errors: i) Insufficient memory ii)Floating exception
e) Compile Errors: i) parse error ii)implicit declaration iii) no matching function iv)Unsatisfied symbols v)incomplete type vi)cannot call member function vii)bad argument viii)cannot allocate an object

**Topic: Components of C language, Standard I/O in C, Fundamental data types, Variables and memory locations**

## ELEMENTS OF C

Every language has some basic elements & grammatical rules. Before starting with programming, we should be acquainted with the basic elements that build the language.

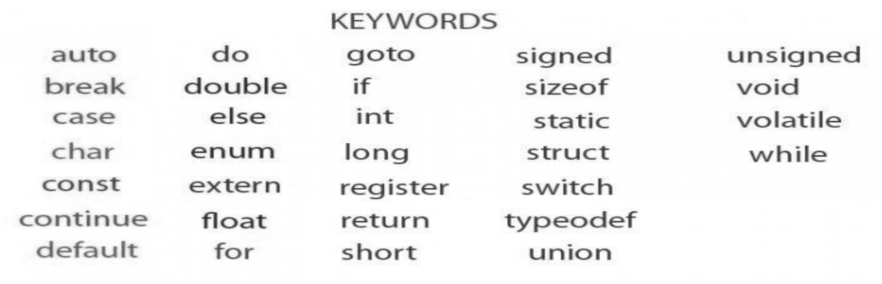
### Character Set

Communicating with a computer involves speaking the language the computer understands. In C, various characters have been given to communicate. Character set in C consists of;

| **Types** | **Character Set** |
|---|---|
| Lower case | a-z |
| Upper case | A-Z |
| Digits | 0-9 |
| Special Character | !@#$%^&* |
| White space | Tab or new lines or space |

### Keywords

Keywords are the words whose meaning has already been explained to the C compiler. 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. There are only 32 keywords available in C. Below figure gives a list of these keywords for your ready reference.



KEYWORDS

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

### Identifier

In the programming language C, an identifier is a combination of alphanumeric characters, the first being a letter of the alphabet or an underline, and the remaining being any letter of the alphabet, any numeric digit, or the underline.

Two rules must be kept in mind when naming identifiers.

1. The case of alphabetic characters is significant. Using "INDEX" for a variable is not the same as using "index" and neither of them is the same as using "InDeX" for a variable. All three refer to different variables.
2. As C is defined, up to 32 significant characters can be used and will be considered significant by most compilers. If more than 32 are used, they will be ignored by the compiler.

**Data Type**

In the C programming language, data types refer to a domain of allowed values & the operations that can be performed on those values. The type of a variable determines how much space it occupies in storage and how the bit pattern stored is interpreted. There are 4 fundamental data types in C, which are- char, int, float &, double. Char is used to store any single character; int is used to store any integer value, float is used to store any single precision floating point number & double is used to store any double precision floating point number.

We can use 2 qualifiers with these basic types to get more types.

There are 2 types of qualifiers-

Sign qualifier- signed & unsigned

Size qualifier- short & long

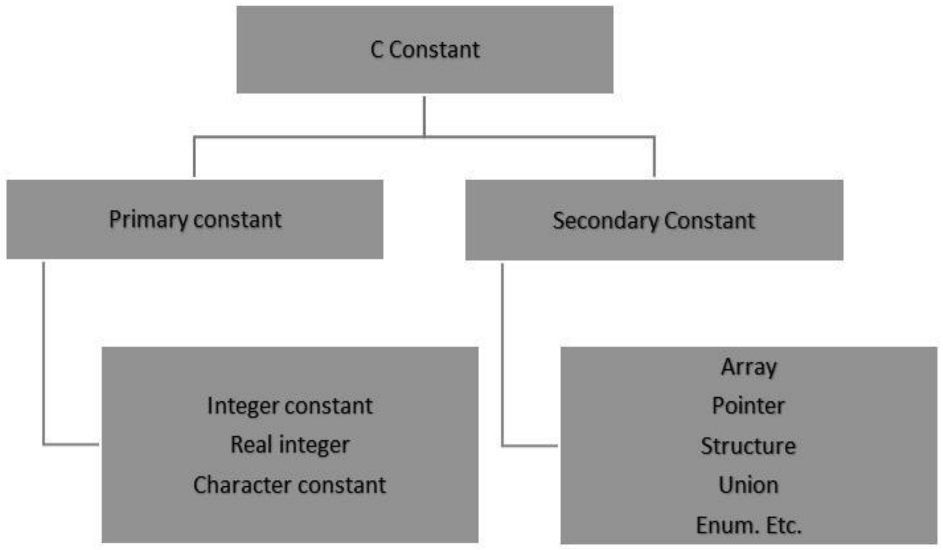
The data types in C can be classified as follows:

| Type | Storage size | Value range |
|---|---|---|
| Char | 1 byte | -128 to 127 |
| unsigned char | 1 byte | 0 to 255 |
| Integrity | 2 or 4 bytes | 32,768 to 32,767 or -2,147,483,648 to 2,147,483,647 |
| nsigned integrity | 2 or 4 bytes | 0 to 65,535 or 0 to 4,294,967,295 |
| Short | 2 bytes | -32,768 to 32,767 |
| unsigned short | 2 bytes | 0 to 65,535 |
| long | 4 bytes | -2,147,483,648 to 2,147,483,64 |
| unsigned long | 4 bytes | 0 to 4,294,967,295 |

| Type | Storage size | Value range | Precision |
|---|---|---|---|
| float | 4 bytes | 1.2E-38 to 3.4E+38 | 6 decimal places |
| double | 8 bytes | 2.3E-308 to 1.7E+308 | 15 decimal places |
| long double | 10 bytes | .4E-4932 to 1.1E+4932 | 19 decimal places |

## Constants

A constant is an entity that doesn't change whereas a variable is an entity that may change constants can be divided into two major categories: Primary Constants, Secondary Constants



**Rules for Constructing Integer Constants:**

- An integer constant must have at least one digit.
- It must not have a decimal point.
- It can be either positive or negative.
- If no sign precedes an integer constant it is assumed to be positive.
- No commas or blanks are allowed within an integer constant
- The allowable range for integer constants is -32768to 32767.

Ex.: 426, +782,-8000, -7605

**Rules for Constructing Real Constants:**

- Real constants are often called Floating Point constants. The real constants could be written in two forms—Fractional form and Exponential form.
- A real constant must have at least one digit.
- It must have a decimal point.
- It could be either positive or negative.
- Default sign is positive.
- No commas or blanks are allowed within a real constant. Ex. +325.34, 426.0, -32.76, -48.5792

**Rules for constructing real constants expressed in exponential form:**

- The mantissa part and the exponential part should be separated by a letter e.
- The mantissa part may have a positive or negative sign.
- Default sign of mantissa part is positive.

- The exponent must have at least one digit, which must be a positive or negative integer. Default sign is positive.
- Range of real constants expressed in exponential form is -3.4e38 to 3.4e38.
- Ex. +3.2e-5, 4.1e8, -0.2e+3, -3.2e-5

**Rules for Constructing Character Constants:**

A character constant is a single alphabet, a single digit or a single special symbol enclosed within single inverted commas.

The maximum length of a character constant can be 1 character.

Ex.: ‘M’, ‘6’, ‘+’

## VARIABLES

Variables are names that are used to store values. It can take different values but one at a time. A data type is associated with each variable & it decides what values the variable can take. Variable declaration requires that you inform C of the variable's name and data type.

Syntax – data type variable name;

Eg:int page_no;

char grade;

float salary;

long y;

**Declaring Variables:**

There are two places where you can declare a variable:

After the opening brace of a block of code (usually at the top of a function)

Before a function name (such as before main() in the program) Consider various examples:

**Initialization of Variables**

When a variable is declared, it contains undefined value commonly known as garbage value. If we want we can assign some initial value to the variables during the declaration itself. This is called initialization of the variable.

Eg-int pageno=10;

**Expressions**

An expression consists of a combination of operators, operands, variables & function calls. An expression can be arithmetic, logical or relational. Here are some expressions:

a+b – arithmetic operation

a>b- relational operation a

== b - logical operation

func (a,b) – function call

**Statements**

Statements are the primary building blocks of a program. A program is a series of statements with some necessary punctuation. A statement is a complete instruction to the computer. In C, statements are indicated by a semicolon at the end. a semicolon is needed to identify instructions that truly are statements.

## INPUT-OUTPUT IN C

When we are saying **Input** that means we feed some data into program. This can be given in the form of file or from command line. C programming language provides a set of built-in functions to read given input and feed it to the program as per requirement.

When we are saying **Output** that means to display some data on screen, printer or in any file. C programming language provides a set of built-in functions to output the data on the computer screen.

Functions printf() and scanf() are the most commonly used to display out and take input respectively. Let us consider an example:

```
#include <stdio.h> //This is needed to run printf() function.
int main()
{ printf("C Programming"); //displays the content inside quotation return 0;
}
Output:C Programming
```

**Explanation:**

Every program starts from main() function.

printf() is a library function to display output which only works if #include<stdio.h>is included at the beginning.

Here, stdio.h is a header file (standard input output header file) and #include is command to paste the code from the header file when necessary. When compiler encounters printf()function and doesn't find stdio.h header file, compiler shows error.

return 0; indicates the successful execution of the  program.

**Input- Output of integers in C**

```
#include<stdio.h>
int main()
{int c=5;
printf("Number=%d",c);
return 0;
}
Output:Number=5
```

**Input - Output of characters**

```
#include <stdio.h>
int main()
{char var1;
printf("Enter character: ");
scanf("%c",&var1);
printf("You entered %c.",var1);
return 0;
}
Output:Enter character: g You entered g.
```

**FORMATTED INPUT-OUTPUT**

Data can be entered & displayed in a particular format. Through format specifications, better presentation of results can be obtained.

**Variations in Output for integer & floats:**

```
#include<stdio.h>
int main()
{printf("Case 1:%6d\n",9876);
/* Prints the number right justified within 6 columns */
printf("Case 2:%3d\n",9876);
/* Prints the number to be right justified to 3 columns but, there are 4 digits so number is not
      right justified */
printf("Case 3:%.2f\n",987.6543);
/* Prints the number rounded to two decimal places */
printf("Case 4:%.f\n",987.6543);
/* Prints the number rounded to 0 decimal place, i.e, rounded to integer */
printf("Case 5:%e\n",987.6543);
/* Prints the number in exponential notation (scientific notation) */
return 0;
}
```

Output

```
Case 1: 9876
Case 2:9876
Case 3:987.65
Case 4:988
```

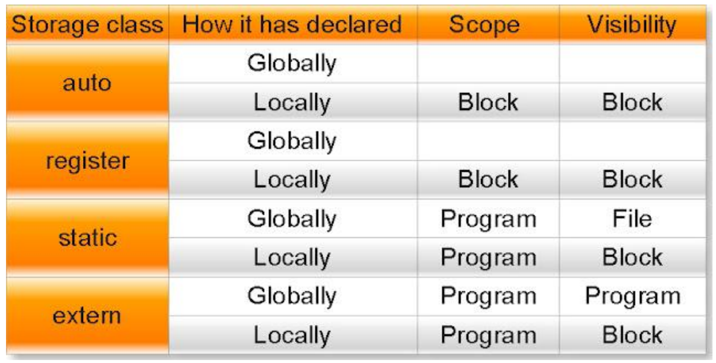
## Topic: Storage classes

- Generally there are two kinds of locations in a computer where such a value can be present, these are Memory and CPU registers. The storage class of a particular variable determines in which of the above two locations the variable's value is stored.
- There are four properties by which storage class of a variable can be recognized. These are scope, default initial value, scope and life.
- A variable's storage class reveals the following things about a variable
  (i) Where the variable is stored.
  (ii) What is the initial value of the variable if the value of the variable is not specified?
  (iii) What is the scope of the variable (To which function or block the variable is available).
  (iv) What is the life of particular variable (Up to what extent the variable exists in a program).

  - In c there are four types of storage class. They are: 1. Auto 2. Register 3. Static 4. Extern
  - Storage class is modifier or qualifier of data types which decides: In which area of memory a particular variable will be stored? What is scope of variable? What is visibility of variable?

**Visibility of a variable in c:** Visibility means accessibility. Up to witch part or area of a program, we can access a variable, that area or part is known as visibility of that variable. For example: In the following figure yellow color represents visibility of variable a.

**Scope of a variable in c**: Meaning of scope is to check either variable is alive or dead. Alive means data of a variable has not destroyed from memory. Up to which part or area of the program a variable is alive, that area or part is known as scope of a variable. There are four type of scope in c:
1. Block scope.
2. Function scope.
3. File scope.
3. Program scope.



| Storage class | How it has declared | Scope | Visibility |
|---|---|---|---|
| auto | Globally | | |
| | Locally | Block | Block |
| register | Globally | | |
| | Locally | Block | Block |
| static | Globally | Program | File |
| | Locally | Program | Block |
| extern | Globally | Program | Program |
| | Locally | Program | Block |

**Automatic Storage Class**
Syntax to declare automatic variable is:
auto datatype variablename;
Example:
auto int i;
Features of Automatic Storage Class are as follows
**Storage:** Memory
**Default Initial Value:** Garbage Value

**Scope:** Local to the block in which the variable is defined
**Life:** Till the control remains within the block in which the variable is defined
by default every variable is automatic variable
The following program illustrates the work of automatic variables.

```
void test();
void main()
{
test();
test();
test();
}
void test()
{
auto int k=10;
printf("%d\n",k);
k++;
}
```

Output:

```
10
10
10
```

In the above program when the function test() is called for the first time ,variable k is created and initialized to 10. When the control returns to main(), k is destroyed. When function test() is called for the second time again k is created , initialized and destroyed after execution of the function. Hence automatic variables came into existence each time the function is executed and destroyed when execution of the function completes.

**Register Storage Class**
Syntax to declare register variable is:
register datatype variablename;
Features of Register Storage Class are as follows:
**Storage:** CPU Registers
**Default Initial Value:** Garbage Value
**Scope:** Local to the block in which the variable is defined
**Life:** Till the control remains within the block in which the variable is defined
For example loop counters are declared as register variables, which are defined as follows:

```
int main()
{
register int a;
for(a=0;i<50000;i++)
printf("%d\t",a);
return 0;
```

In the above program, variable **a** was used frequently as a loop counter so the variable a is defined as a register variable. Register is a not a command but just a request to the compiler to allocate the memory for the variable into the register. If free registers are available than memory will be allocated in to the registers. And if there are no free registers then memory will be allocated in the RAM only (Storage is in memory i.e. it will act as automatic variable).
Every type of variables can be stored in CPU registers. Suppose the microprocessor has 16 bit registers then they can't hold a float or a double value which requires 4 and 8 bytes respectively.

But if you use register storage class for a float or double variable then you will not get any error message rather the compiler will treat the float and double variable as be of automatic storage class(i.e. Will treat them like automatic variables).

**Static Storage Class**

Syntax to declare static variable is:
static datatype variablename;
Example:
static int i;
Features of Static Storage Class are as follows
**Storage:** Memory
**Default Initial Value:** Zero
**Scope:** Local to the block in which the variable is defined
**Life:** Value of the variable continues to exist between different function calls
Now look at the previous program with k is declared as static instead of automatic.

```
void test();
void main()
{
test();
test();
test();
}
void test()
{
static int k=10;
printf("%d\n",k);
k++;
}
```

Output:10 11 12

Here in the above program the output is 10, 11, 12, because if a variable is declared as static then it is initialized only once and that variable will never be initialized again. Here variable k is declared as static, so when test() is called for the first time k value is initialized to 10 and its value is incremented by 1 and becomes 11. Because k is static its value persists. When test() is called for the second time k is not reinitialized to 10 instead its old value 11 is available. So now 11 is get printed and it value is incremented by 1 to become 12. Similarly for the third time when test() is called 12(Old value) is printed and its value becomes 13 when executes the statement 'k++;'

So the main difference between automatic and static variables is that static variables are initialized to zero if not initialized but automatic variables contain an unpredictable value(Garbage Value) if not initialized and static variables does not disappear when the function is no longer active , their value persists, i.e. if the control comes back to the same function again the static variables have the same values they had last time.

General advice is avoid using static variables in a program unless you need them, because their values are kept in memory when the variables are not active which means they occupies space in memory that could otherwise be used by other variables.

**External Storage Class**

Syntax to declare static variable is:

extern datatype variablename;
Example:
extern int i;
Features of External Storage Class are as follows
**Storage:** Memory
**Default Initial Value:** Zero
**Scope:** Global
**Life:** Till the program's execution doesn't come to an end

External variables differ from automatic, register and static variables in the context of scope, external variables are global on the contrary automatic, register and static variables are local. External variables are declared outside all functions, therefore are available to all functions that want to use them.

If the program size is very big then code may be distributed into several files and these files are compiled and object codes are generated. These object codes linked together with the help of linker and generate ".exe" file. In the compilation process if one file is using global variable but it is declared in some other file then it generate error called undefined symbol error. To overcome this we need to specify global variables and global functions with the keyword extern before using them into any file. If the global variable is declared in the middle of the program then we will get undefined symbol error, so in that case we have to specify its prototype using the keyword extern.

So if a variable is to be used by many functions and in different files can be declared as external variables. The value of an uninitialized external variable is zero. The declaration of an external variable declares the type and name of the variable, while the definition reserves storage for the variable as well as behaves as a declaration. The keyword **extern** is specified in declaration but not in definition. Now look at the following four statements

1. auto int a;
2. register int b;
3. static int c;
4. extern int d;

Out of the above statements first three are definitions where as the last one is a declaration. Suppose there are two files and their names are File1.c and File2.c respectively. Their contents are as follows:

**File1.c**
```
int n=10;
void hello()
{
printf("Hello");
}
```
**File2.c**
```
extern int n;
extern void hello();
void main()
{printf("%d", n);
hello();}
```
In the above program File1.obj+ File2.obj=File2.exe and in File2.c value of n will be 10.

## Module – 2: (Arithmetic expressions & Conditional Branching)

**Topic: Arithmetic expressions and precedence: Operators and expression using numeric and relational operators, mixed operands, type conversion, logical operators, bit operations, assignment operator, operator precedence and associability.**

**OPERATORS-**An operator is a symbol that tells the compiler to perform specific mathematical or logical manipulations. C language is rich in built-in operators and provides the following types of operators: Arithmetic Operators, Relational Operators, Logical Operators, Bitwise Operators, Assignment Operators, Increment and decrement operators, Conditional operators, Misc Operators

**Arithmetic operator:** Assume variable A holds 10 and variable B holds 20 then:

| Operator | Description | Example |
|---|---|---|
| + | Adds two operands | A + B will give 30 |
| - | Subtracts second operand from the first | A – B will give -10 |
| * | Multiplies both operands | A * B will give 200 |
| / | Divides numerator by de-numerator | B / A will give 2 |
| % | Modulus Operator and remainder of after an integer Division | B % A will give 0 |
| ++ | Increments operator increases integer value by one | A++ will give 11 |
| -- | Decrements operator decreases integer value by one | A–will give 9 |

**Relational Operators:**

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

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

**Logical Operators:**

Assume variable A holds 1 and variable B holds 0, then:

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

**Bitwise Operators**

Bitwise operator works on bits and performs bit-by-bit operation. These operators can operate upon int and char but not on float and double..Bit wise operators in C language are; & (bitwise AND), | (bitwise OR), ~ (bitwise OR), ^ (XOR), << (left shift) and >> (right shift).The truth tables for &, |, and ^ are as follows:

| **P** | **Q** | **p & q** | **p \| q** | **p ^ q** |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 | 1 |
| 1 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 | 1 |

Assume variable A holds 60 (00111100) and variable B holds 13 (00001101), then:

| **Operator** | **Description** | **Example** |
|---|---|---|
| & | Binary AND Operator copies a bit to the result if it exists in both operands. | (A & B) will give 12, which is 0000 1100 |
| \| | Binary OR Operator copies a bit if it exists in either operand. | (A \| B) will give 61, which is 0011 1101 |
| ^ | Binary XOR Operator copies the bit if it is set in one operand but not both. | (A ^ B) will give 49, which is 0011 0001 |
| ~ | Binary Ones Complement Operator is unary and has the effect of 'flipping' bits. | (~A ) will give -61, which is 1100 0011 in 2's complement form. |
| << | Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand. | A << 2 will give 240 which is 1111 0000 |
| >> | Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand. | A >> 2 will give 15 which is 0000 1111 |

**Assignment Operators:**
In C programs, values for the variables are assigned using assignment operators.
There are following assignment operators supported by C language:

| Operator | Description | Example |
|---|---|---|
| = | nple assignment operator, Assigns values from right side operands to left side operand | C = A + B will assign value of A + B into C |
| += | ld AND assignment operator, It adds right operand to the left operand and assign the result to left operand | C += A is equivalent to C = C + A |
| -= | Subtract AND assignment operator, It subtracts right pperand from the left operand and assign the result to left Operand | C -= A is equivalent to C = C – A |
| *= | Multiply AND assignment operator, It multiplies right operand with the left operand and assign the result to left Operand | C *= A is equivalent to C = C * A |
| /= | Divide AND assignment operator, It divides left operand ith the right operand and assign the result to left operand | C /= A is equivalent to C = C / A |
| %= | odulus AND assignment operator, It takes modulus using two operands and assign the result to left operand | C %= A is equivalent to C = C % A |
| <<= | Left shift AND assignment operator | C <<= 2 is same as C = C << 2 |
| >>= | Right shift AND assignment operator | C >>= 2 is same as C = C >> 2 |
| &= | Bitwise AND assignment operator | C &= 2 is same as C = C & 2 |
| ^= | bitwise exclusive OR and assignment operator | C ^= 2 is same as C = C ^ 2 |
| \|= | bitwise inclusive OR and assignment operator | C \|= 2 is same as C = C \| 2 |

**Increment/Decrement OPERATOR**

In C, ++ and – are called increment and decrement operators respectively. Both of these operators are unary operators, i.e, used on single operand. ++ adds 1 to operand and – subtracts 1 to operand respectively. For example:

```
Let a=5 and b=10
a++;                    //a becomes 6
a--;//a becomes 5
++a;                    //a becomes 6
--a;//a becomes 5
```

When i++ is used as prefix(like: ++var), ++var will increment the value of var and then return it but, if ++ is used as postfix(like: var++), operator will return the value of operand first and then only increment it. This can be demonstrated by an example:

```
#include <stdio.h>
int main()
{int c=2,d=2;
printf("%d\n",c++); //this statement displays 2 then, only c incremented by 1 to 3.
Printf("%d",++c);            //this statement increments 1 to c then, only c is displayed.
Return 0;
}
Output:2 4
```

**Conditional Operators (? :)**

Conditional operators are used in decision making in C programming, i.e, executes different statements according to test condition whether it is either true or false.

**Syntax of conditional operators;**

conditional_expression?expression1:expression2

If the test condition is true (that is, if its value is non-zero), expression1 is returned and if false expression2 is returned.

y = ( x> 5 ? 3 : 4 ) ;➔this statement will store 3 in y if x is greater than 5, otherwise it will store 4 in y.

**Misc Operators:**

There are few other operators supported by c language.

| **Operator** | **Description** | **Example** |
|---|---|---|
| sizeof() | It is a unary operator which is used in finding the size of data type, constant, arrays, structure etc. | sizeof(a), where a is integer, will return 4. |
| & | Returns the address of a variable. | &a; will give actual address of the variable. |
| * | Pointer to a variable. | *a; will pointer to a variable. |

**Operators Precedence in C**

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

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

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

| **Category** | **Operator** | **Associativity** |
|---|---|---|
| Postfix | () [] -> . ++ - - | Left to right |
| Unary | + - ! ~ ++ - - (type)* &sizeof | Right to left |
| Multiplicative | * / % | Left to right |
| Additive | + - | Left to right |
| Shift | <<>> | Left to right |
| Relational | <<= >>= | Left to right |
| Equality | == != | Left to right |
| Bitwise AND | & | Left to right |
| Bitwise XOR | ^ | Left to right |
| Bitwise OR | \| | Left to right |
| Logical AND | && | Left to right |
| Logical OR | \|\| | Left to right |
| Conditional | ?: | Right to left |
| Assignment | = += -= *= /= %=>>= <<= &= ^= \|= | Right to left |
| Comma | , | Left to right |

**Type casting**

- Type casting /type conversion is a way to convert a variable from one data type to another data type. For example, if you want to store a long value into a simple integer then you can typecast long to int.
- You can convert values from one type to another explicitly using the cast operator. There are two types of type casting in c languages that are implicit conversions and Explicit Conversions.
- New data type should be mentioned before the variable name or value in brackets which to be typecast.

**C TYPE CASTING EXAMPLE PROGRAM:**

In the below C program, 7/5 alone will produce integer value as 1.So, type cast is done before division to retain float value (1.4).

```
#include <stdio.h>
int main ()
{
   float x;
   x = (float) 7/5;
   printf("%f",x);
}
```

Output:

```
    1.400000
```

**WHAT IS TYPE CASTING IN C LANGUAGE?**

Converting an expression of a given type into another type is known as type casting. It is best practice to convert lower data type to higher data type to avoid data loss.Data will be truncated when the higher data type is converted to lower. For example, if a float is converted to int, data which is present after the decimal point will be lost.

There are two types of type casting in c language.

**TYPES OF TYPECASTING IN C**

Types of type casting in C Programming

Implicit Conversion

Explicit Conversion

**1. IMPLICIT CONVERSION**

Implicit conversions do not require any operator for converted. They are automatically performed when a value is copied to a compatible type in the program.

```
#include<stdio.h>
#include<conio.h>
void main()
{
    int i=20;
    double p;
    clrscr();
   p=i; // implicit conversion
    printf("implicit value is %d",p);
    getch();}
```

Output:-

```
    implicit value is 20.
```

**2. EXPLICIT CONVERSION**

In C language, Many conversions, especially those that imply a different interpretation of the value, require an explicit conversion.

Example :-

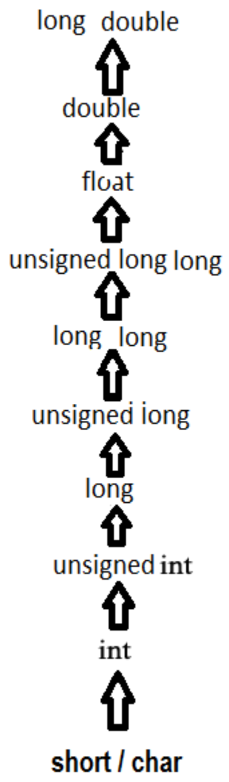
```
#include<stdio.h>
#include<conio.h>
void main()
{
    int i=20;
     short p;
     clrscr();
     p = (short) i; // Explicit conversion
     printf("Explicit value is %d",p);
     getch();
}
```

Output:-

```
     Explicit value is 20.
```

**Usual Arithmetic Conversion**

The usual arithmetic conversions are implicitly performed to cast their values in a common type, C uses the rule that in all expressions except assignments, any implicit type conversions made from a lower size type to a higher size type as shown below:



Type Casting In C Language

**DIFFERENCE BETWEEN TYPE CASTING AND TYPE CONVERSION**

| BASIS FOR COMPARISON | TYPE CASTING | TYPE CONVERSION |
|---|---|---|
| Definition | When a user can convert the one data type into other then it is called as the type casting. | Type Conversion is that which automatically converts the one data type into another. |
| Implemented | Implemented on two ‘incompatible’ data types. | Implemented only when two data types are ‘compatible’. |
| Operator | For casting a data type to another, a casting operator ‘()’ is required. | No operator required. |
| Implemented | It is done during program designing. | It is done explicitly while compiling. |
| Conversion type | Narrowing conversion. | Widening conversion. |
| Example | int x; byte y; …y= (byte) x; | int x=3; float y; y=a; // value in y=3.000. |

**Inbuilt Typecast Functions In C:**

There are many inbuilt type casting functions available in C language which performs data type conversion from one type to another.

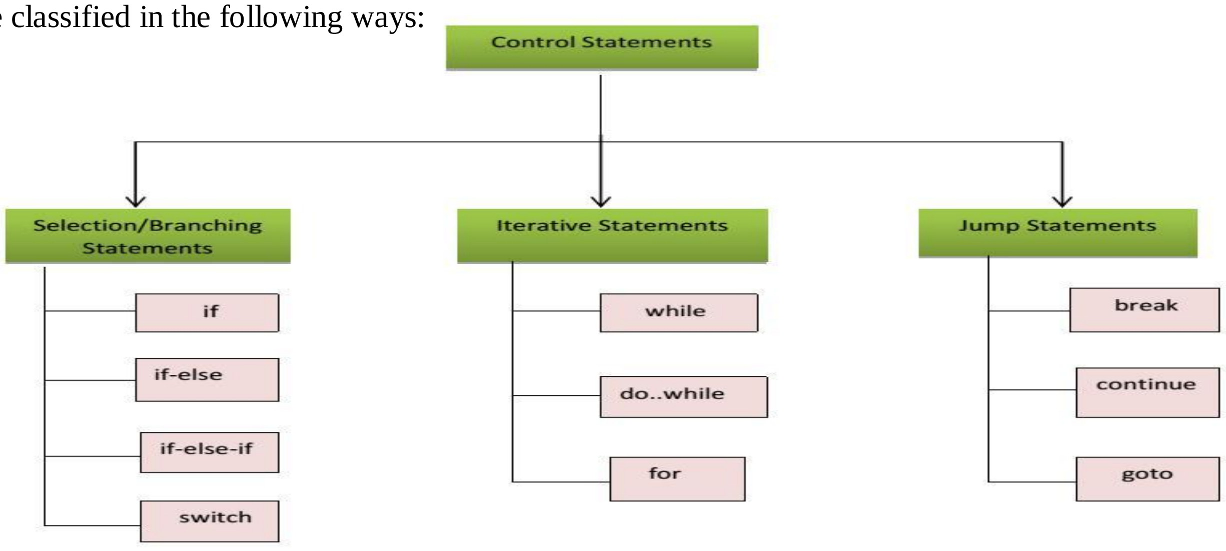
| **S.No** | **Typecast Function** | **Description** |
|---|---|---|
| 1 | atof() | Convert string to Float |
| 2 | atoi() | Convert string to int |
| 3 | atol() | Convert string to long |
| 4 | itoa() | Convert int to string |
| 5 | ltoa() | Convert long to string |

**Topic: Conditional Branching: Applying if and switch statements, nesting if and else, use of break and Default with switch.**

**If, if-else, Case switch statements**

**CONTROL STATEMENTS**

Control statements enable us to specify the order in which the various instructions in the program are to be executed. They define how the control is transferred to other parts of the program. Control statements are classified in the following ways:



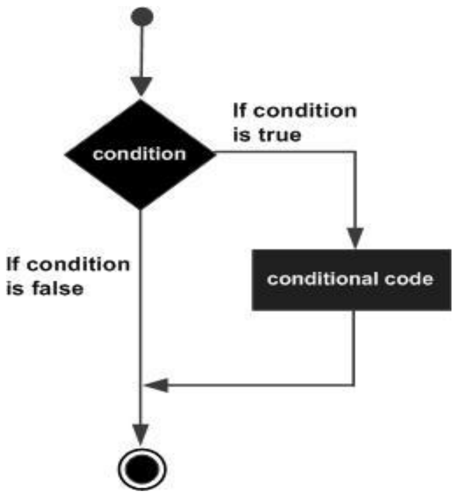
**a) If statement**

Syntax:

```
        if(boolean_expression)
         {  /* statement(s) will execute if the Boolean expression is true */
         }
```

If the Boolean expression evaluates to true then the block of code inside the if statement will be executed. If boolean expression evaluates to false then the first set of code after the end of the if statement (after the closing curly brace) will be executed. C programming language assumes any non-zero and non-null values as true and if it is either zero or null then it is assumed as false value.

**Flow Diagram:**



**Example:**

```
#include <stdio.h>
 int main ()
{       int a = 10;
        if( a < 20 )
             {   Printf("a is less than 20\n" );
             }
        Printf("value of a is : %d\n", a);
 }
```

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

```
a is less than 20;
value of a is : 10
```

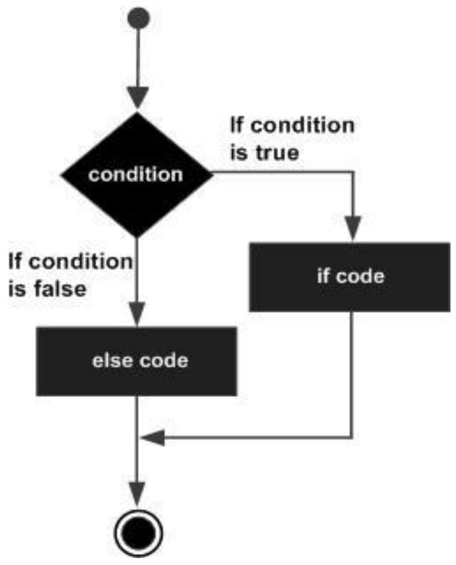
**(b) if –else statement**

Syntax: The syntax of an if...else statement in C programming language is:

```
if(boolean_expression)
{   /* statement(s) will execute if the boolean expression is true */
}
else
{  /* statement(s) will execute if the boolean expression is false */
}
```

If the Boolean expression evaluates to true then the if block of code will be executed otherwise else block of code will be executed programming language assumes any non-zero and non-null values as true and if it is either zero or null then it is assumed as false value.

**Flow Diagram:**



**Example:**

```
#include <stdio.h>
 main ()
{   int a = 100;
     if( a < 20 )
   {      Printf("a is less than 20\n" );
    }
   else
   {      Printf("a is not less than 20\n" );
    }  Printf("value of a is : %d\n", a);}
```

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

```
a is not less than 20;
value of a is : 100
```

**(c) Nested if-else statement**

The syntax for a nested if statement is as follows:

```
if( boolean_expression 1)
{   /* Executes when the Boolean expression 1 is true */
   if(boolean_expression 2)
   {     /* Executes when the Boolean expression 2 is true */
   }
}
```

**Example:**

```
#include <stdio.h>
main ()
{   int a = 100;
    int b = 200;
   if( a == 100 )
   {            if( b == 200 )
         {  printf("Value of a is 100 and b is 200\n" );
      }
   }
   printf("Exact value of a is : %d\n", a );
   printf("Exact value of b is : %d\n", b );
    return 0;
}
```

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

```
Value of a is 100 and b is 200
Exact value of a is : 100
Exact value of b is : 200
```

**(d) A switch statement**

Allows a variable to be tested for equality against a list of values. Each value is called a case, and the variable being switched on is checked for each switch case. Syntax:

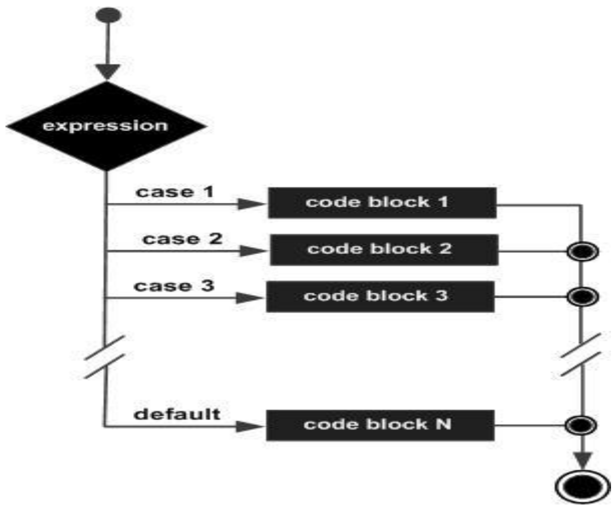
```
 switch(expression)
{    case constant-expression  :
       statement(s);
       break; /* optional */
    case constant-expression  :
       statement(s);
       break; /* optional */

    /* you can have any number of case statements */
    default : /* Optional */
       statement(s);
}
```

**The following rules apply to a switch statement**:

- You can have any number of case statements within a switch. Each case is followed by the value to be compared to and a colon.

- The constant-expression for a case must be the same data type as the variable in the switch, and it must be a constant or a literal.
- When the variable being switched on is equal to a case, the statements following that case will execute until a break statement is reached.
- When a break statement is reached, the switch terminates, and the flow of control jumps to the next line following the switch statement.
- Not every case needs to contain a break. If no break appears, the flow of control will fall through to subsequent cases until a break is reached.
- A switch statement can have an optional default case, which must appear at the end of the switch. The default case can be used for performing a task when none of the cases is true. No break is needed in the default case.

**Flow Diagram:**



**Example:**

```
main ()
{  char grade = 'B';
     switch(grade)
   {   case 'A' :
     printf("Excellent!\n" );
     break;
   case 'B' :
   case 'C' :
     printf("Well done\n" );
     break;
   case 'D' :
     printf("You passed\n" );
     break;
   case 'F' :
     printf("Better try again\n" );
     break;
   default :
     printf("Invalid grade\n" );
   }   printf("Your grade is  %c\n", grade );
 }
```

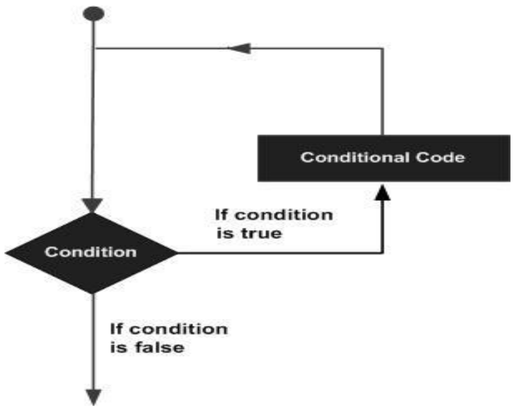
When the above code is compiled and executed, it produces following result:

```
Well done
Your grade is B
```

**Module – 3: (Loops & Functions)**

**Topic: Iteration and loops: use of while, do while and for loops, multiple loop variables, use of break and**

**Program Loops and Iteration** A loop statement allows us to execute a statement or group of statements multiple times and following is the general from of a loop statement in most of the programming languages:



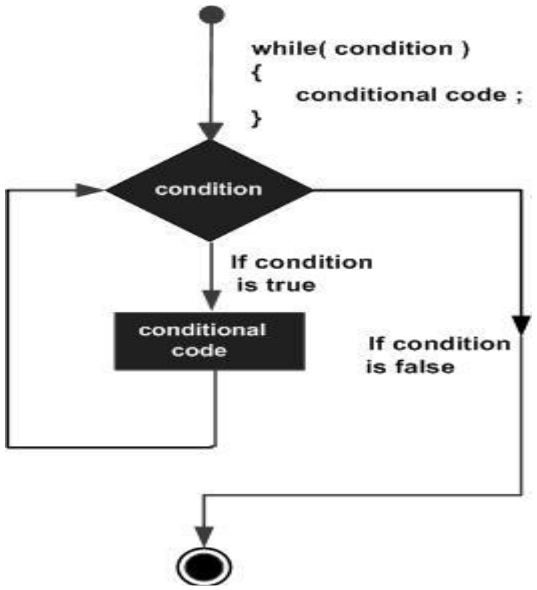
**(a) Uses of while loop**

A while loop statement in C programming language repeatedly executes a target statement as long as a given condition is true.

Syntax:

```
while(condition)
{   statement(s);
}
```

Here statement(s) may be a single statement or a block of statements. The condition may be any expression, and true is any nonzero value. The loop iterates while the condition is true.When the condition becomes false, program control passes to the line immediately following the loop. **Flow Diagram:**



**Example:**

```
#include <stdio.h>
 main ()
{      int a = 10;
    while( a < 20 )
   {
      printf("value of a: %d\n", a);
      a++;
   }
 }
```

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

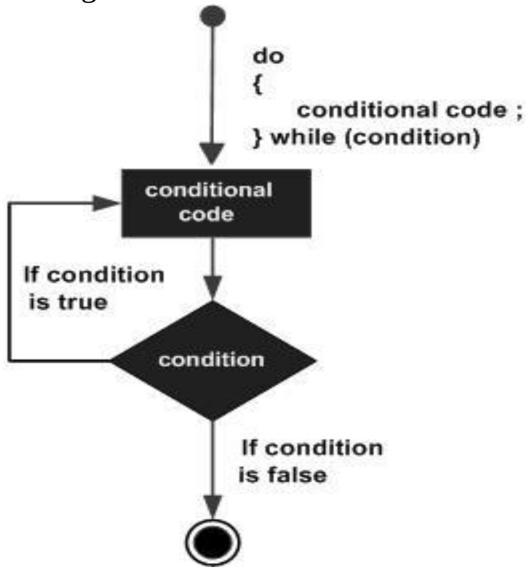
```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 15
value of a: 16
value of a: 17
value of a: 18
value of a: 19
```

**(b) Do-while loop**

Unlike for and while loops, which test the loop condition at the top of the loop, the do...while loop in C programming language checks its condition at the bottom of the loop. A do...while loop is similar to a while loop, except that a do...while loop is guaranteed to execute at least one time. **Syntax:**

```
do
{   Statement;
} while ( condition );
```

If the condition is true, the flow of control jumps back up to do, and the statement(s) in the loop execute again. This process repeats until the given condition becomes false. **Flow Diagram:**



**Example:**

```
#include <stdio.h>
 main ()
{   int a = 10;
  do
   {       printf("value of a: %d\n", a);
       a = a + 1;
   } while ( a < 20 );
 }
```

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

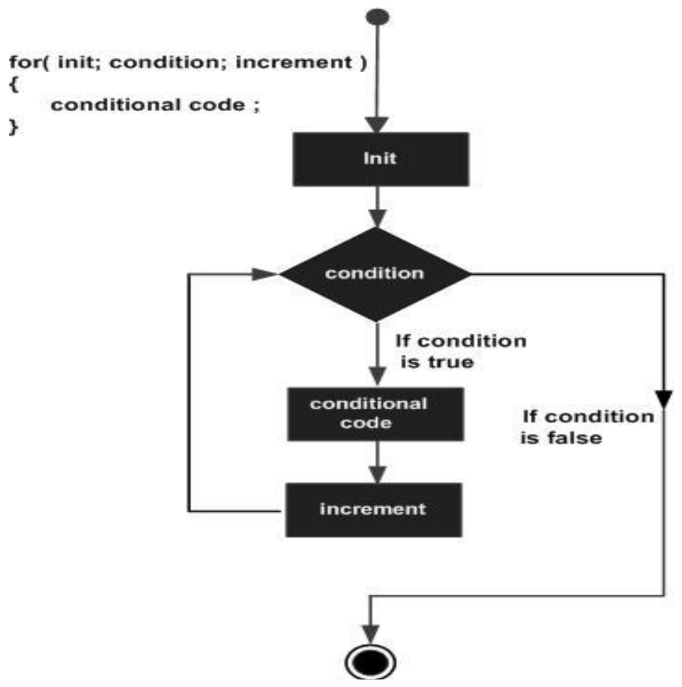
```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 15
value of a: 16
value of a: 17
value of a: 18
value of a: 19
```

**(c) For loops**

for loop is a repetition control structure that allows you to efficiently write a loop that needs to execute a specific number of times. **Syntax:**

```
for ( initialization ; condition; increment )
{   statement(s);
}
```

**Flow Diagram:**



**Example:**

```
#include <stdio.h>
 main ()
{   for( int a = 10; a < 20; a = a + 1 )
  {      printf("value of a: %d\n", a);
  }
}
```

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

```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 15
value of a: 16
value of a: 17
value of a: 18
value of a: 19
```

## Jump statements

### Loop Control Statements

Loop control statements change execution from its normal sequence. When execution leaves a scope, all automatic objects that were created in that scope are destroyed. C supports the following control statements.

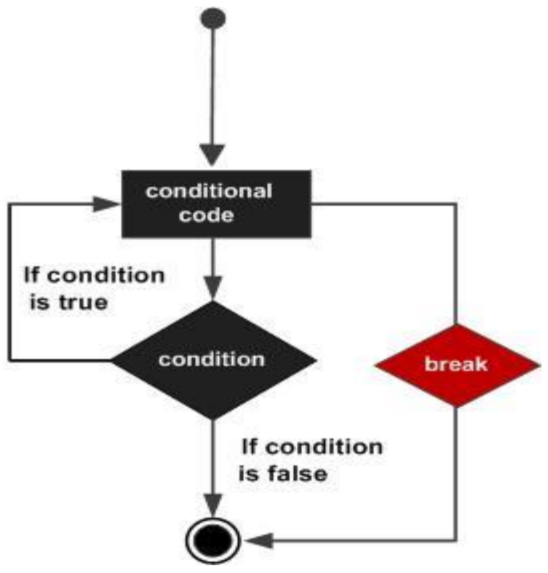
**(i) *Break statement*:** Terminates the loop or switch statement and transfers execution to the statement immediately following the loop or switch. The break statement in C programming language has following two usage:

1. When the break statement is encountered inside a loop, the loop is immediately terminated and program control resumes at the next statement following the loop.
2. It can be used to terminate a case in the switch statement.

**Syntax:**

Break;

**Flow Diagram:**



**Example:**

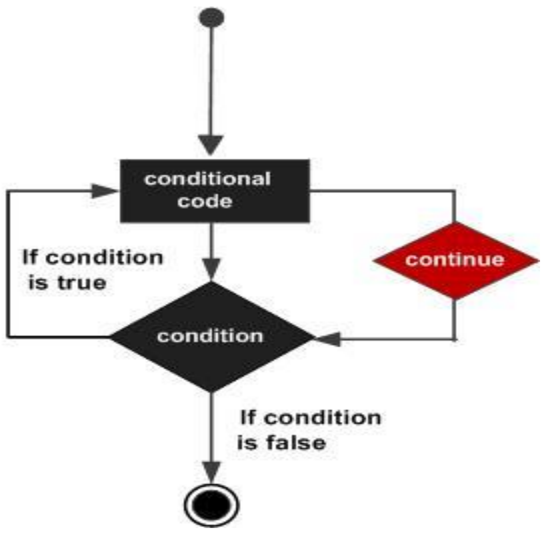
```
#include <stdio.h>
 main ()
{   int a = 10;
    while( a < 20 )
   {    printf("value of a: %d\n", a);
        a++;
        if( a > 15)
      {      break;
      }
   }
 }
```

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

```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 15
```

**(ii)** ***Continue statement*****:** Causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating. The continue statement in C programming language works somewhat like the break statement. Instead of forcing termination, however, continue forces the next iteration of the loop to take place, skipping any code in between. **Syntax:** continue;

**Flow Diagram:**



Example:

```
#include <stdio.h>
main ()
{   int a = 10;
      do
      {  if( a == 15)
         {       a = a + 1;
                continue;
```

```
          }
      printf("value of a: %d\n", a);
      a++;
        } while( a < 20 );
 }
```

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

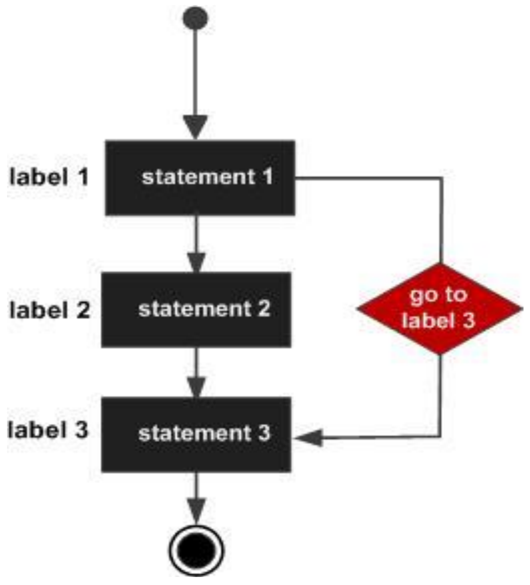
```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 16
value of a: 17
value of a: 18
value of a: 19
```

**(iii) Goto Statement:** Transfers control to the labeled statement. Though it is not advised to use goto statement in your program A goto statement in C programming language provides an unconditional jump from the goto to a labeled statement in the same function.

**Syntax:**

```
goto label;
label: statement;
```

**Flow Diagram:**



**Example:**

```
#include <stdio.h>
main ()
{   int a = 10;
   LOOP: do
              {                    if( a == 15)
                  {         a = a + 1;
                   goto LOOP;
                  }
```

```
      printf("value of a: %d\n", a);
      a++;
        }while( a < 20 );
 }
```
When the above code is compiled and executed, it produces following result:
```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 16
value of a: 17
value of a: 18
value of a: 19
```

**Topic: Functions- Introduction, types of functions, functions with array, passing parameters to functions, Call by value, call by reference**

**FUNCTION**

**MONOLITHIC VS MODULAR PROGRAMMING:**

1. Monolithic Programming indicates the program which contains a single function for the large program.
2. Modular programming help the programmer to divide the whole program into different modules and each module is separately developed and tested. Then the linker will link all these modules to form the complete program.
3. On the other hand monolithic programming will not divide the program and it is a single thread of execution. When the program size increases it leads inconvenience and difficult to maintain.

**Disadvantages of monolithic programming:** 1. Difficult to check error on large programs. 2. Difficult to maintain. 3. Code can be specific to a particular problem. i.e. it cannot be reused.

**Advantage of modular programming:** 1. Modular program are easier to code and debug. 2. Reduces the programming size. 3. Code can be reused in other programs. 4. Problem can be isolated to specific module so easier to find the error and correct it.

**FUNCTION:**

A function is a group of statements that together perform a task. Every C program has at least one function, which is **main()**, and all the most trivial programs can define additional functions.

**Function Declaration OR Function Prototype:**

**1.** It is also known as function prototype.

**2.** It inform the computer about the three things
   a) Name of the function
   b) Number and type of arguments received by the function.
   c) Type of value return by the function

Syntax:
return type function name (type1 arg1 , type2 arg2);
OR
return type function name (type1 type2);

**3.** Calling function need information about called function .If called function is place before calling function then the declaration is not needed.

**Function Definition:**

1. It consists of code description and code of a function. It consists of two parts
   a) Function header
   b) Function coding

Function definition tells what the I/O functions are and what is going to do.
Syntax:
```
return type function name (type1 arg1 , type2 arg2)
{local variable; statements ;
return (expression);
}
```

2. Function definition can be placed anywhere in the program but generally placed after the main function .
3. Local variable declared inside the function is local to that function. It cannot be used anywhere in the program and its existence is only within the function.
4. Function definition cannot be nested.
5. Return type denote the type of value that function will return and return type is optional if omitted it is assumed to be integer by default.

**USER DEFINE FUNCTIONS VS STANDARD FUNCTION:**

**User Define Function:**

A function that is declare, calling and define by the user is called user define function. Every user define function has three parts as:

1. Prototype or Declaration
2. Calling
3. Definition

**FUNCTION CATAGORIES**

There are four main categories of the functions these are as follows:

1. Function with no arguments and no return values.
2. Function with no arguments and a return value.
3. Function with arguments and no return values.
4. Function with arguments and return values.

**Function with no arguments and no return values:**

syntax:

```
void funct (void); main ( )
{
funct ( );
}
void funct ( void );
{
}
```

**Function with no arguments and a return value:**

This type of functions has no arguments but a return value

**example:**

```
int msg (void) ; int main ( )
{
int s = msg ( );
printf( “summation = %d” , s);
}
int msg ( void )
{
int a, b, sum ; sum = a+b ; return (sum) ;
}
```

**Function with arguments and no return values:**

Example:

```
void msg ( int , int );
int main ( )
```

```
{
int a,b;
a= 2; b=3; msg( a, b);
}
void msg ( int a , int b)
{
int s ;
sum = a+b;
printf (“sum = %d” , s ) ;
}
```

**Function with arguments and return value:**

Here calling function of arguments that passed to the called function and called function return value to calling function.

**example:**

```
int  msg ( int , int ) ;
int main ( )
{
int a, b;
a= 2; b=3;
int s = msg (a, b);
printf (“sum = %d” , s ) ;
}
int msg( int a , int b)
{
int sum ;
sum =a+b ; return (sum);
}
```

## ACTUAL ARGUMENTS AND FORMAL ARGUMENTS

**Actual Arguments:**

1. Arguments which are mentioned in the function in the function call are known as calling function.
2. These are the values which are actual arguments called to the function.

**Formal Arguments:**

1. Arguments which are mentioned in function definition are called dummy or formal argument.
2. These arguments are used to just hold the value that is sent by calling function.
3. Formal arguments are like other local variables of the function which are created when function call starts and destroyed when end function.

Basic difference between formal and local argument are:

a) Formal arguments are declared within the ( ) where as local variables are declared at beginning.

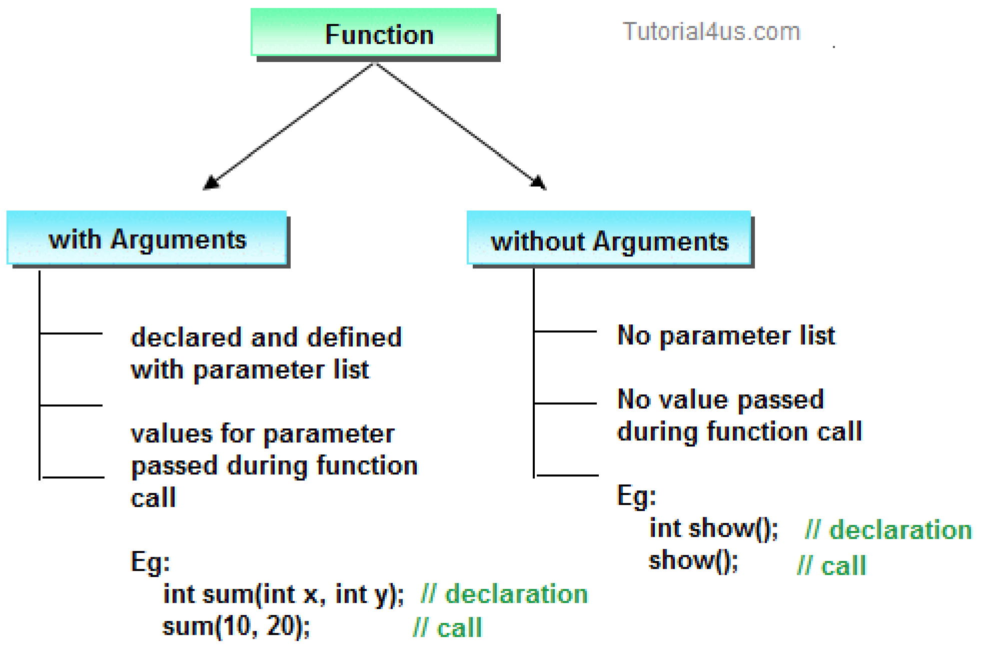
b) Formal arguments are automatically initialized when a value of actual argument is passed.

c) Where other local variables are assigned variable through the statement inside the function body.

## PARAMETER PASSING TECHNIQUES:

### Call by Value and Call by Reference in C

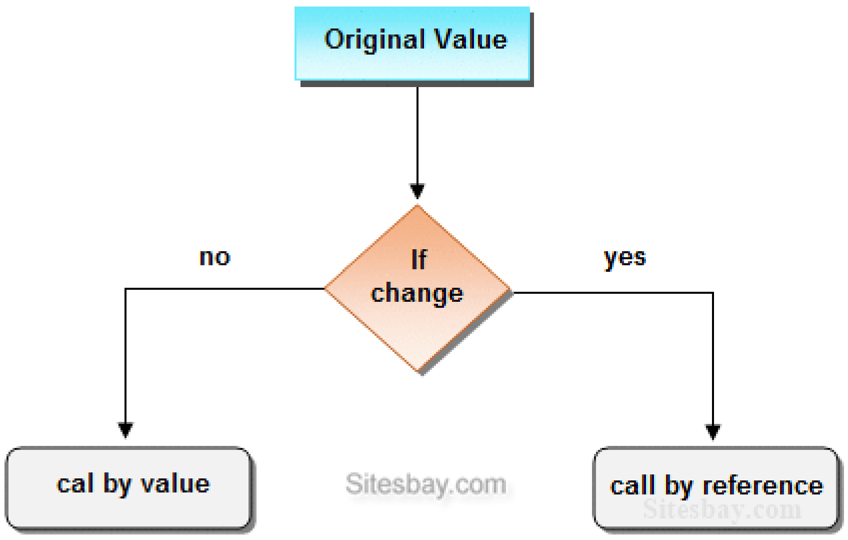
On the basis of arguments there are two types of function are available in C language, they are;


Function
Tutorial4us.com
with Arguments
without Arguments
declared and defined with parameter list
values for parameter passed during function call
Eg:
int sum(int x, int y); // declaration
sum(10, 20); // call
No parameter list
No value passed during function call
Eg:
int show(); // declaration
show(); // call

- With argument
- Without argument

If a function takes any arguments, it must declare variables that accept the values as a arguments. These variables are called the formal parameters of the function. There are two ways to pass value or data to function in C language which is given below;

- call by value
- call by reference



### Call by value

In call by value, **original value can not be changed** or modified. In call by value, when you passed value to the function it is locally stored by the function parameter in stack memory location. If you change the value of function parameter, it is changed for the current function only but it not change the value of variable inside the caller method such as main().

**Call by value**

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

void swap(int a, int b)
{
 int temp;
 temp=a;
 a=b;
 b=temp;
}

void main()
{
 int a=100, b=200;
 clrscr();
 swap(a, b);  // passing value to function
 printf("\nValue of a: %d",a);
 printf("\nValue of b: %d",b);
 getch();
}
```

**Output**

```
Value of a: 200
Value of b: 100
```

**Call by reference**

In call by reference, **original value is changed** or modified because we pass reference (address). Here, address of the value is passed in the function, so actual and formal arguments shares the same address space. Hence, any value changed inside the function, is reflected inside as well as outside the function.

**Example Call by reference**

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

void swap(int *a, int *b)
{
 int temp;
 temp=*a;
 *a=*b;
 *b=temp;
}

void main()
{
 int a=100, b=200;
 clrscr();
 swap(&a, &b);  // passing value to function
 printf("\nValue of a: %d",a);
 printf("\nValue of b: %d",b);
 getch();
}
```

**Output**

```
Value of a: 200
Value of b: 100
```

**Difference between call by value and call by reference.**

| call by value | call by reference |
|---|---|
| This method copy original value into function as a arguments. | This method copy address of arguments into function as a arguments. |
| Changes made to the parameter inside the function have no effect on the argument. | Changes made to the parameter affect the argument. Because address is used to access the actual argument. |
| Actual and formal arguments will be created in different memory location | Actual and formal arguments will be created in same memory location |

## Topic: Recursion

When Function is call within same function is called **Recursion**. The function which calls same function is called **recursive function**. In other word when a function calls itself then that function is called **Recursive function**.

Recursive function are very useful to solve many mathematical problems like to calculate factorial of a number, generating Fibonacci series, etc.

**Advantage of Recursion**

- Function calling related information will be maintained by recursion.
- Stack evaluation will be take place by using recursion.
- In fix prefix, post-fix notation will be evaluated by using recursion.

**Disadvantage of Recursion**

- It is a very slow process due to stack overlapping.
- Recursive programs can create stack overflow.
- Recursive functions can create as loops.

**Find the Factorial of any number using recursion**

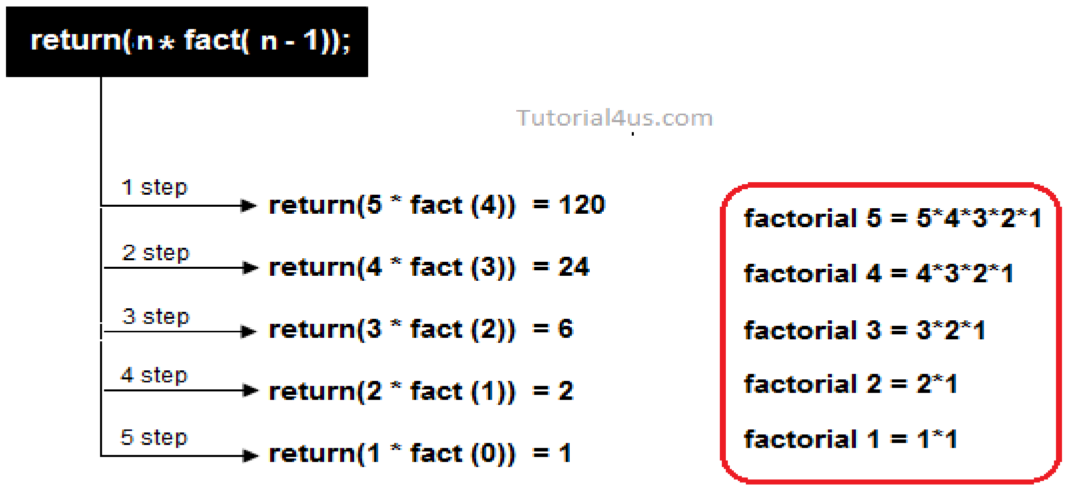
**Example**

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

void main()
{
int fact(int);
int i,f,num;
clrscr();
printf("Enter any number: ");
scanf("%d",&num);
f=fact(num);
printf("Factorial: %d",f);
getch();
}

int fact(int n)
{
if(a<0)
return(-1);
if(a==0)
return(1);
else
{
return(n*fact(n-1));
}}
```

**Output**

```
Enter any number: 5
```

```
Factorial: 120
```



**Find the Table of any number using recursion**

**Example**

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

void main()
{
int table(int,int);
int n,i;      // local variable
clrscr();
printf("Enter any num : ");
scanf("%d",&n);
for(i=1;i< =10;i++)
{
printf(" %d*%d= %d\n",n,i,table(n,i));
}
getch();
}
int table(n,i)
{
int t;
if(i==1)
{
return(n);
}
else
{
t=(table(n,i-1)+n);
return(t);
//return(table(n,i-1)+n);
}
}
Output
```

```
Enter any number: 5
5*1= 5
5*2= 10
5*3= 15
5*4= 20
5*5= 25
5*6= 30
5*7= 35
5*8= 40
5*9= 45
5*10= 50
```

**Examples of Recursion**

Q1. Write a program using recursion to find the summation of numbers from 1 to n.

**Ans:** We can say 'sum of numbers from 1 to n can be represented as sum of numbers from 1 to n-1 plus n' i.e.

Sum of numbers from 1 to n = n + Sum of numbers from 1 to n-1
= n + n-1 + Sum of numbers from 1 to n-2
= n+ n-1 + n-2 + ............... +1

The program which implements the above logic is as follows:

```
#include<stdio.h>
void main()
{
int n,s;
printf("Enter a number");
scanf("%d",&n);
s=sum(n);
printf("Sum of numbers from 1 to %d is %d",n,s);
}
int sum(int m) int r;
if(m==1)
return (1);
else
r=m+sum(m-1);/*Recursive Call*/
return r;}Output:Enter a number 5 15
```

## Module – 4: (Arrays & Basic Algorithms)

**Topic: Arrays- Array notation and representation, manipulating array elements, using multi dimensional arrays, passing arrays to functions.**

### Introduction

- A data structure is the way data is stored in the machine and the functions used to access that data. An easy way to think of a data structure is a collection of related data items.
- An array is a data structure that is a collection of variables of one type that are accessed through a common name.
- Each element of an array is given a number by which we can access that element which is called an index. It solves the problem of storing a large number of values and manipulating them.
- In an Array values of same type are stored. An array is a group of memory locations related by the fact that they all have the same name and same type. To refer to a particular location or element in the array we specify the name to the array and position number of particular element in the array.

### One Dimensional Array

**Declaration:**

Before using the array in the program it must be declared
Syntax:
data_type array_name[size];

data_type represents the type of elements present in the array. array_name represents the name of the array.
Size represents the number of elements that can be stored in the array.

Example:
int age[100];
float sal[15];
char grade[20];

Here age is an integer type array, which can store 100 elements of integer type. The array sal is floating type array of size 15, can hold float values. Grade is a character type array which holds 20 characters.

**Initialization:**

We can explicitly initialize arrays at the time of declaration. Syntax:
data_type array_name[size]={value1, value2,……..valueN};

Value1, value2, valueN are the constant values known as initializers, which are assigned to the array elements one after another.

Example:
int marks[5]={10,2,0,23,4};

The values of the array elements after this initialization are: marks[0]=10, marks[1]=2, marks[2]=0, marks[3]=23, marks[4]=4

NOTE:

1. In 1-D arrays it is optional to specify the size of the array. If size is omitted during initialization then the compiler assumes the size of array equal to the number of initializers. Example:int marks[]={10,2,0,23,4};Here the size of array marks is initialized to 5.
2. We can't copy the elements of one array to another array by simply assigning it.Example:

```
int a[5]={9,8,7,6,5}; int b[5];
b=a;                                    //not valid
```
we have to copy all the elements by using for loop.
```
                for(a=i; i<5; i++) b[i]=a[i];
```

**Processing:**
For processing arrays we mostly use for loop. The total no. of passes is equal to the no. of elements present in the array and in each pass one element is processed.
Example:
```
#include<stdio.h>
main()
{
int a[3],i;
for(i=0;i<=2;i++)                                    //Reading the array values
{
printf("enter the elements");
scanf("%d",&a[i]);
}
for(i=0;i<=2;i++)                                //display the array values
{
printf("%d",a[i]);
printf("\n");
}
}
```
This program reads and displays 3 elements of integer type.

**TWO DIMENSIONAL ARRAYS**
Arrays that we have considered up to now are one dimensional array, a single line of elements. Often data come naturally in the form of a table, e.g. spreadsheet, which need a two-dimensional array.
**Declaration:**
The syntax is same as for 1-D array but here 2 subscripts are used.
Syntax:
data_type array_name[rowsize][columnsize];
Rowsize specifies the no.of rows
Columnsize specifies the no.of columns.

Example:
int a[4][5];
This is a 2-D array of 4 rows and 5 columns. Here the first element of the array is a[0][0] and last element of the array is a[3][4] and total no.of elements is 4*5=20.

**Initialization:**
2-D arrays can be initialized in a way similar to 1-D arrays.
Example:
int m[4][3]={1,2,3,4,5,6,7,8,9,10,11,12};
The values are assigned as follows:
The initialization of group of elements as follows:
int m[4][3]={{11},{12,13},{14,15,16},{17}};
The values are assigned as:
Note:
In 2-D arrays it is optional to specify the first dimension but the second dimension should always be present.
Example: int m[][3]={
{1,10},
{2,20,200},
{3}, {4,40,400} };
Here the first dimension is taken 4 since there are 4 roes in the initialization list. A 2-D array is known as matrix.

**Processing:**
For processing of 2-D arrays we need two nested for loops. The outer loop indicates the rows and the inner loop indicates the columns.
Example:
int a[4][5];

```
                Reading values in a
 for(i=0;i<4;i++)
for(j=0;j<5;j++)
scanf("%d",&a[i][j]);
                Displaying values of a
                for(i=0;i<4;i++)
for(j=0;j<5;j++)
printf("%d",a[i][j]);
```

Examples:1.**Matrix addition in c language**

```
      #include<stdio.h>
    int main()
    {  int a[3][3],b[3][3],c[3][3],i,j;
      printf("Enter the First matrix->");
      for(i=0;i<3;i++)
          for(j=0;j<3;j++)
              scanf("%d",&a[i][j]);
      printf("\nEnter the Second matrix->");
      for(i=0;i<3;i++)
          for(j=0;j<3;j++)
              scanf("%d",&b[i][j]);
      printf("\nThe First matrix is\n");
      for(i=0;i<3;i++)
    {      printf("\n");
          for(j=0;j<3;j++)
```

```
         printf("%d\t",a[i][j]);
  }
  printf("\nThe Second matrix is\n");
  for(i=0;i<3;i++)
{      printf("\n");
      for(j=0;j<3;j++)
      printf("%d\t",b[i][j]);
   }
   for(i=0;i<3;i++)
       for(j=0;j<3;j++)
            c[i][j]=a[i][j]+b[i][j];
   printf("\nThe Addition of two matrix is\n");
   for(i=0;i<3;i++)
{       printf("\n");
       for(j=0;j<3;j++)
            printf("%d\t",c[i][j]);
   }
   return 0;
}
```

**Algorithm:Addition of two matrixes:**

Rule: Addition of two matrixes is only possible if both matrixes are of same size. Suppose two matrixes A and B is of same size m X n

**Sum of two marixes is defined as**

(A + B)ij = Aij + Bij

Where $1 \le i \le m$ and $1 \le j \le n$

For example:

Suppose two matrixes A and B of size of 2 X 3 is as follow:

$$A = \begin{pmatrix} 5 & 10 & 20 \\ 8 & 6 & 5 \end{pmatrix} \quad B = \begin{pmatrix} 3 & 8 & 5 \\ 2 & 9 & 3 \end{pmatrix}$$

Addition of two matrixes:

$$A + B = \begin{pmatrix} 5+3 & 10+8 & 20+5 \\ 8+2 & 6+9 & 5+3 \end{pmatrix} = \begin{pmatrix} 8 & 18 & 25 \\ 10 & 15 & 8 \end{pmatrix}$$

**2.C code for matrix multiplication**

```
#include<stdio.h>
int main()
{  int a[5][5],b[5][5],c[5][5],i,j,k,sum=0,m,n,o,p;
  printf("\nEnter the row and column of first matrix");
  scanf("%d %d",&m,&n);
  printf("\nEnter the row and column of second matrix");
  scanf("%d %d",&o,&p);
  if(n!=o)
{      printf("Matrix mutiplication is not possible");
      printf("\nColumn of first matrix must be same as row of second matrix");
  }
  else{
      printf("\nEnter the First matrix->");
      for(i=0;i<m;i++)
      for(j=0;j<n;j++)
          scanf("%d",&a[i][j]);
      printf("\nEnter the Second matrix->");
      for(i=0;i<o;i++)
      for(j=0;j<p;j++)
          scanf("%d",&b[i][j]);
      printf("\nThe First matrix is\n");
      for(i=0;i<m;i++)
{       printf("\n");
      for(j=0;j<n;j++)
{           printf("%d\t",a[i][j]);
      }
      }
      printf("\nThe Second matrix is\n");
      for(i=0;i<o;i++)
{       printf("\n");
      for(j=0;j<p;j++)
{           printf("%d\t",b[i][j]);
      }
      }
      for(i=0;i<m;i++)
      for(j=0;j<p;j++)
          c[i][j]=0;
      for(i=0;i<m;i++)
{ //row of first matrix
      for(j=0;j<p;j++)
{  //column of second matrix
          sum=0;
          for(k=0;k<n;k++)
              sum=sum+a[i][k]*b[k][j];
          c[i][j]=sum;
      }
      }
  }
  printf("\nThe multiplication of two matrix is\n");
  for(i=0;i<m;i++)
{       printf("\n");
      for(j=0;j<p;j++)
{           printf("%d\t",c[i][j]);
      }
  }
  return 0;
}
```

**Algorithm:Multiplication of two matrixes:**

Rule: Multiplication of two matrixes is only possible if first matrix has size m X **n** and other matrix has size **n** x r. Where m, n and r are any positive integer. **Multiplication of two matrixes is defined as**

$$[AB]_{i,j} = \sum_{s=1}^{n} A_{i,s}B_{s,j}$$

Where $1 \le i \le m$ and $1 \le j \le n$

For example:

Suppose two matrixes A and B of size of 2 x 2 and 2 x 3 respectively:

$$A = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \quad B = \begin{pmatrix} 5 & 6 & 7 \\ 8 & 9 & 10 \end{pmatrix}$$

Multiplication of two matrixes:

$$A * B = \begin{pmatrix} 1*5 + 2*8 & 1*6 + 2*9 & 1*7 + 2*10 \\ 3*5 + 4*8 & 3*6 + 4*9 & 3*7 + 4*10 \end{pmatrix}$$

$$A * B = \begin{pmatrix} 21 & 24 & 27 \\ 47 & 54 & 61 \end{pmatrix}$$

**3.Sum of diagonal elements of a matrix in c**

```
#include<stdio.h>
int main()
{  int a[10][10],i,j,sum=0,m,n;
  printf("\nEnter the row and column of matrix: ");
  scanf("%d %d",&m,&n);
  printf("\nEnter the elements of matrix: ");
  for(i=0;i<m;i++)
      for(j=0;j<n;j++)
           scanf("%d",&a[i][j]);
  printf("\nThe matrix is\n");
  for(i=0;i<m;i++)
{      printf("\n");
      for(j=0;j<m;j++)
{      printf("%d\t",a[i][j]);
      }
 }
 for(i=0;i<m;i++)
{     for(j=0;j<n;j++)
{          if(i==j)
             sum=sum+a[i][j];
     }
 }
 printf("\n\nSum of the diagonal elements of a matrix is: %d",sum);
 return 0;
}
```

**Sample output:**

```
Enter the row and column of matrix: 3 3
Enter the elements of matrix: 2
```

```
3
5
6
7
9
2
6
7
The matrix is
2       3       5
6       7       9
2       6       7
Sum of the diagonal elements of a matrix is: 16
```

**Algorithm:Sum of diagonal element of matrix:**

$$A = \begin{pmatrix} \mathbf{A_{11}} & A_{12} & A_{13} \\ A_{21} & \mathbf{A_{22}} & A_{23} \\ A_{31} & A_{32} & \mathbf{A_{33}} \end{pmatrix}$$

Diagonal elements have been shown in the bold letter.We can observer the properties any element Aij will diagonal element if and only if i = j

**4.C program to find the largest element in an array**

```
#include<stdio.h>
int main()
{  int a[50],size,i,big;
  printf("\nEnter the size of the array: ");
  scanf("%d",&size);
  printf("\nEnter %d elements in to the array: ”, size);
  for(i=0;i<size;i++)
      scanf("%d",&a[i]);
  big=a[0];
  for(i=1;i<size;i++)
{      if(big<a[i])
           big=a[i];
  }
  printf("\nBiggest: %d",big);
  return 0;
}
```

**1-d arrays using functions**

**Passing individual array elements to a function**

We can pass individual array elements as arguments to a function like other simple variables.
Example:

```
#include<stdio.h> void check(int); void main()
{
int a[10],i; clrscr();
printf("\n enter the array elements:"); for(i=0;i<10;i++)
{
scanf("%d",&a[i]);
check(a[i]);
}
void check(int num)
{
if(num%2==0)
printf("%d is even\n",num); else
printf("%d is odd\n",num);
}
```

Output:

```
enter the array elements:
1 2 3 4 5 6 7 8 9 10
    1           is odd
    2           is even
    3           is odd
    4           is even
    5           is odd
    6           is even
    7           is odd
    8           is even
    9           is odd
    10          is even
```

Example:
C program to pass a single element of an array to function

```
#include <stdio.h> void display(int a)
{
printf("%d",a);
}
int main()
{
int c[]={2,3,4};
display(c[2]); //Passing array element c[2] only. return 0;
}
```

Output

```
2 3 4
```

**Passing whole 1-D array to a function**

We can pass whole array as an actual argument to a function the corresponding formal arguments should be declared as an array variable of the same type.

Example:
```
#include<stdio.h>
main()
{
int i, a[6]={1,2,3,4,5,6};
func(a);
printf("contents of array:");
for(i=0;i<6;i++)
printf("%d",a[i]);
printf("\n");
}
func(int val[])
{
int sum=0,i;
for(i=0;i<6;i++)
{
val[i]=val[i]*val[i];
sum+=val[i];
}
printf("the sum of squares:%d", sum);
}
```

Output
```
contents of array: 1 2 3 4 5 6
the sum of squares: 91
```

**Example.2:**

Write a C program to pass an array containing age of person to a function. This function should find average age and display the average age in main function.
```
#include <stdio.h>
float average(float a[]);
int main()
{
float avg, c[]={23.4, 55, 22.6, 3, 40.5, 18};
avg=average(c);                    /* Only name of array is passed as argument. */
printf("Average age=%.2f",avg);
return 0;
}
float average(float a[])
{
int i;
float avg, sum=0.0;
for(i=0;i<6;++i)
{
sum+=a[i];
}
avg =(sum/6);
return avg;
}
```

Output

```
Average age= 27.08
```

Example: 2

C Program to Print the Alternate Elements in an Array

```
#include <stdio.h> void main()
{
int array[10]; int i, j, temp;
printf("enter the element of an array \n"); for (i = 0; i < 10; i++)
scanf("%d", &array[i]);
printf("Alternate elements of a given array \n"); for (i = 0; i < 10; i += 2)
printf( "%d\n", array[i]) ;
}
```

## SEARCHING

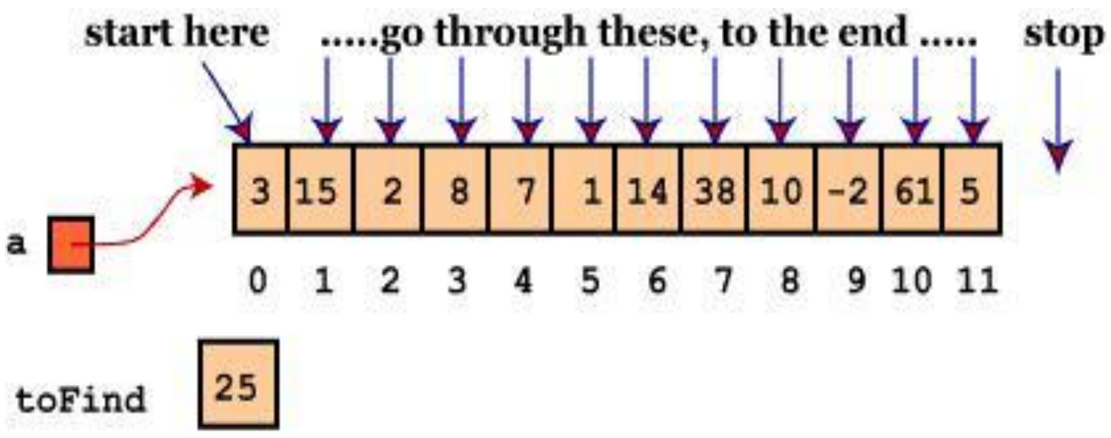
### Linear Search

Linear search or sequential search is a method for finding a particular value in a list that consists of checking every one of its elements, one at a time and in sequence, until the desired one is found.

Linear search is the simplest search algorithm; it is a special case of brute-force search.

### <u>How Linear Search works</u>

Linear search in an array is usually programmed by stepping up an index variable until it reaches the last index. This normally requires two comparisons for each list item: one to check whether the index has reached the end of the array, and another one to check whether the item has the desired value.



### <u>Linear Search Algorithm</u>

```
1.      Repeat For J = 1 to N
2.      If (ITEM == A[J]) Then
3.      Print: ITEM found at location J
4. Return    [End of If]
[End of For Loop]
5. If (J > N) Then
6.      Print: ITEM doesn't
exist [End of If]
7.      Exit
```

```
//CODE
Main()
{int a[10],i,n,m,c=0, x;
printf("Enter the size of an array: ");
scanf("%d",&n);
printf("Enter the elements of the array: ");
for(i=0;i<=n-1;i++){
scanf("%d",&a[i]);
}
printf("Enter the number to be search: ");
scanf("%d",&m);
for(i=0;i<=n-1;i++){
if(a[i]==m){
x=I;
c=1;
break;
}
}
if(c==0)
printf("The number is not in the list");
else
printf("The number is found at location %d", x);
}
```

**Complexity of linear Search**

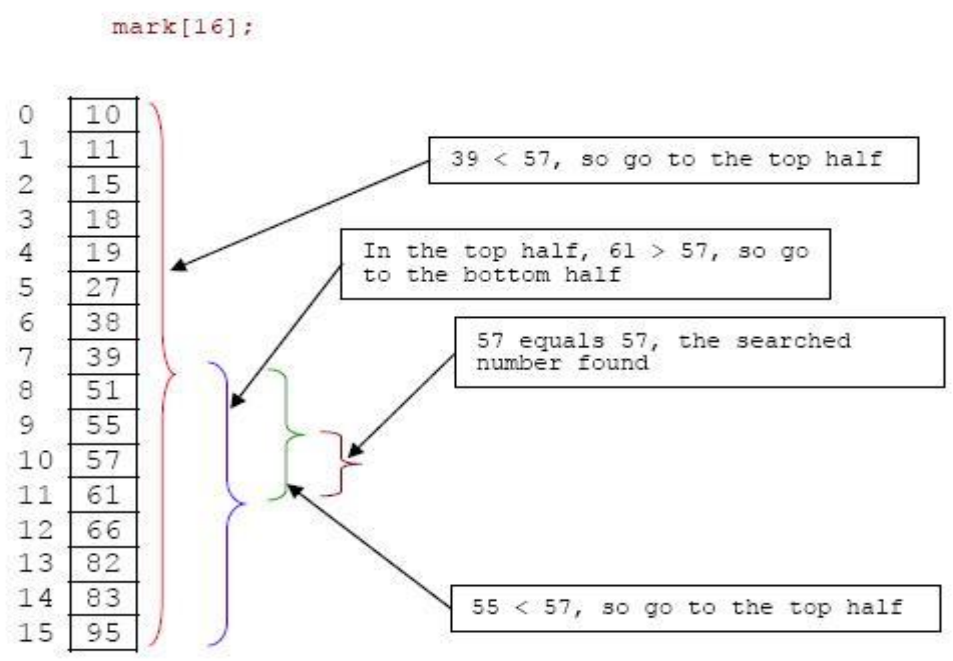
Linear search on a list of n elements. In the worst case, the search must visit every element once. This happens when the value being searched for is either the last element in the list, or is not in the list. However, on average, assuming the value searched for is in the list and each list element is equally likely to be the value searched for, the search visits only n/2 elements. In best case the array is already sorted i.e O(1)

**Binary Search**

A binary search or half-interval search algorithm finds the position of a specified input value (the search "key") within an array sorted by key value. For binary search, the array should be arranged in ascending or descending order. In each step, the algorithm compares the search key value with the key value of the middle element of the array. If the keys match, then a matching element has been found and its index is returned. Otherwise, if the search key is less than the middle element's key, then the algorithm repeats its action on the sub-array to the left of the middle element or, if the search key is greater, on the sub-array to the right. If the remaining array to be searched is empty, then the key cannot be found in the array and a special "not found" indication is returned.

**<u>How Binary Search Works</u>**

Searching a sorted collection is a common task. A dictionary is a sorted list of word definitions. Given a word, one can find its definition. A telephone book is a sorted list of people's names, addresses, and telephone numbers. Knowing someone's name allows one to quickly find their telephone number and address.


mark[16];
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
10 11 15 18 19 27 38 39 51 55 57 61 66 82 83 95
39 < 57, so go to the top half
In the top half, 61 > 57, so go to the bottom half
57 equals 57, the searched number found
55 < 57, so go to the top half

**Binary Search Algorithm**

```
1.      Set BEG = 1 and END = N
2.      Set MID = (BEG + END) / 2
3.      Repeat step 4 to 8 While (BEG <= END) and (A[MID] ≠ ITEM)
4.      If (ITEM < A[MID]) Then
5.      Set END = MID – 1
6.      Else
7.      Set BEG = MID + 1
[End of If]
8.      Set MID = (BEG + END) / 2
9.      If (A[MID] == ITEM) Then
10.     Print: ITEM exists at location MID
11.     Else
12.     Print: ITEM doesn’t exist
[End of If]
13.     E
xit
```

```
Main()
{
int ar[10],val,mid,low,high,size,i;
clrscr();
printf("\nenter the no.s of elements u wanna input in array\n");
scanf("%d",&size);
for(i=0;i<size;i++)
{
printf("input the element no %d\n",i+1);
scanf("%d",&ar[i]);
}
```

```
printf("the arry inputed is \n");
for(i=0;i<size;i++)
{printf("%d\t",ar[i]);
}
low=0;
high=size-1;
printf("\ninput the no. u wanna search \n");
scanf("%d",&val);
while(val!=ar[mid]&&high>=low)
{
mid=(low+high)/2;
if(ar[mid]==val)
{
printf("value found at %d position",mid+1);
}
if(val>ar[mid]){
low=mid+1;
}
else
{
high=mid-1;
}}
```

## INTRODUCTION TO SORTING

Sorting is nothing but storage of data in sorted order, it can be in ascending or descending order. The term Sorting comes into picture with the term Searching. There are so many things in our real life that we need to search, like a particular record in database, roll numbers in merit list, a particular telephone number, any particular page in a book etc.

### Sorting Efficiency

There are many techniques for sorting. Implementation of particular sorting technique depends upon situation. Sorting techniques mainly depends on two parameters.

First parameter is the execution time of program, which means time taken for execution of program.

Second is the space, which means space taken by the program.

## TYPES OF SORTING

• An internal sort is any data sorting process that takes place entirely within the main memory of a computer. This is possible whenever the data to be sorted is small enough to all be held in the main memory.

• External sorting is a term for a class of sorting algorithms that can handle massive amounts of data. External sorting is required when the data being sorted do not fit into the main memory of a computing device (usually RAM) and instead they must reside in

the slower external memory (usually a hard drive). External sorting typically uses a hybrid sort-merge strategy. In the sorting phase, chunks of data small enough to fit in main memory are read, sorted, and written out to a temporary file. In the merge phase, the sorted sub files are combined into a single larger file.

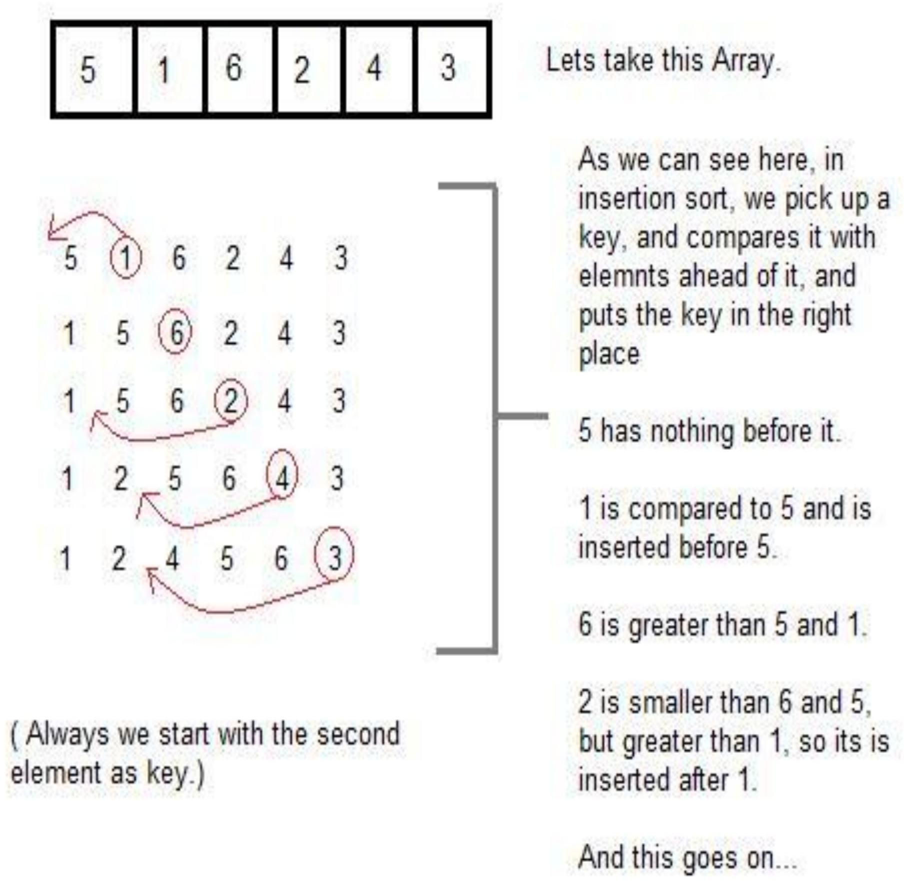
• We can say a sorting algorithm is stable if two objects with equal keys appear in the same order in sorted output as they appear in the input unsorted array.

**Insertion sort**

It is a simple sorting algorithm that builds the final sorted array (or list) one item at a time. This algorithm is less efficient on large lists than more advanced algorithms such as quicksort, heapsort, or merge sort. However, insertion sort provides several advantages:

· Simple implementation
· Efficient for small data sets
· Stable; i.e., does not change the relative order of elements with equal keys
· In-place; i.e., only requires a constant amount O(1) of additional memory space.

**How Insertion Sort Works**


5 1 6 2 4 3
Lets take this Array.
5 1 6 2 4 3
1 5 6 2 4 3
1 5 6 2 4 3
1 2 5 6 4 3
1 2 4 5 6 3
As we can see here, in insertion sort, we pick up a key, and compares it with elemnts ahead of it, and puts the key in the right place
5 has nothing before it.
1 is compared to 5 and is inserted before 5.
6 is greater than 5 and 1.
2 is smaller than 6 and 5, but greater than 1, so its is inserted after 1.
And this goes on...
( Always we start with the second element as key.)

**<u>Insertion Sort Algorithm</u>**

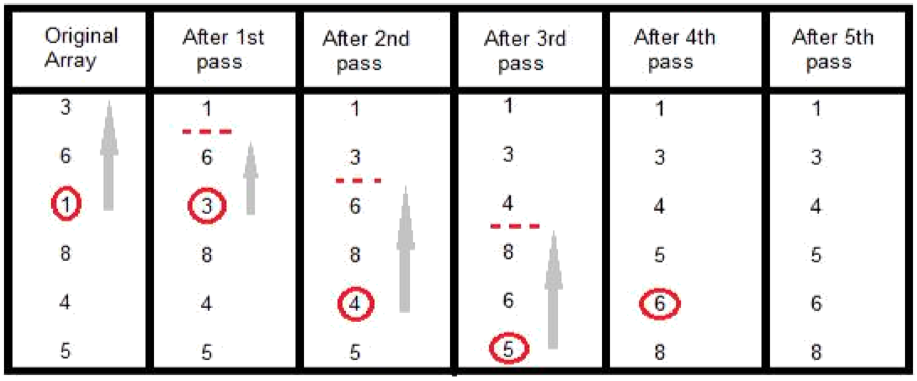
This algorithm sorts the array A with N elements.

```
1.      Set A[0]=-12345(infinity i.e. Any large no)
2.      Repeat step 3 to 5 for k=2 to n
3.      Set key=A[k] And j=k-1
4.      Repeat while key<A[j]
A)  Set A[j+1]=A[j]
b) j=j-1
5.      Set A[j+1]=key
6.      Return
```

```
//CODE

IS()
{int A[6] = {5, 1, 6, 2, 4, 3};
int i, j, key;
for(i=1; i<6; i++)
{
key = A[i];
j = i-1;
while(j>=0 && key < A[j])
{
A[j+1] = A[j];
j--;
}
A[j+1] = key;
}
```

## Selection Sort

Selection sorting is conceptually the simplest sorting algorithm. This algorithm first finds the smallest element in the array and exchanges it with the element in the first position, then find the second smallest element and exchange it with the element in the second position, and continues in this way until the entire array is sorted



| Original Array | After 1st pass | After 2nd pass | After 3rd pass | After 4th pass | After 5th pass |
|---|---|---|---|---|---|
| 3 | 1 | 1 | 1 | 1 | 1 |
| 6 | 6 | 3 | 3 | 3 | 3 |
| 1 | 3 | 6 | 4 | 4 | 4 |
| 8 | 8 | 8 | 8 | 5 | 5 |
| 4 | 4 | 4 | 6 | 6 | 6 |
| 5 | 5 | 5 | 5 | 8 | 8 |

**<u>How Selection Sort works</u>**

In the first pass, the smallest element found is 1, so it is placed at the first position, then leaving first element, smallest element is searched from the rest of the elements, 3 is the smallest, so it is then placed at the second position. Then we leave 1 and 3, from the rest of the elements, we search for the smallest and put it at third position and keep doing this, until array is sorted

**Selection Sort Algorithm**

```
1.Repeat For J = 0 to N-1
2.Set MIN = J
3.Repeat For K = J+1 to N
4.If (A[K] < A[MIN]) Then
5.Set MIN = K
6.Interchange A[J] and
A[MIN] [End of Step 1 For
Loop]
7.Exit
```

//CODE

```
void selectionSort(int a[], int size)
{
int i, j, min, temp;
for(i=0; i < size-1; i++ )
{
min = i;  //setting min as i
for(j=i+1; j < size; j++)
{
if(a[j] < a[min])  //if element at j is less than element at min position
{
min = j;         //then set min as j
}
}
temp = a[i];
a[i] = a[min];
a[min] = temp;
}
}
```

**Bubble Sort**

Bubble Sort is an algorithm which is used to sort N elements that are given in a memory for eg: an Array with N number of elements. Bubble Sort compares all the element one by one and sort them based on their values.

It is called Bubble sort, because with each iteration the smaller element in the list bubbles up towards the first place, just like a water bubble rises up to the water surface.

Sorting takes place by stepping through all the data items one-by-one in pairs and comparing adjacent data items and swapping each pair that is out of order.

**How Bubble Sort Works**

Let us take the array of numbers "5 1 4 2 8", and sort the array from lowest number to greatest number using bubble sort. In each step, elements written in bold are being compared. Three passes will be required.

First Pass:
( 5 1 4 2 8 ) →( 1 5 4 2 8 ), Here, algorithm compares the first two elements, and swaps since 5 > 1.
( 1 5 4 2 8 ) →( 1 4 5 2 8 ), Swap since 5 > 4
( 1 4 5 2 8 ) →( 1 4 2 5 8 ), Swap since 5 > 2
( 1 4 2 5 8 ) →( 1 4 2 5 8 ), Now, since these elements are already in order (8 > 5), algorithm does not swap them.

Second Pass:
( 1 4 2 5 8 ) →( 1 4 2 5 8 )
( 1 4 2 5 8 ) →( 1 2 4 5 8 ), Swap since 4 > 2
( 1 2 4 5 8 ) →( 1 2 4 5 8 )
( 1 2 4 5 8 ) →( 1 2 4 5 8 )
Now, the array is already sorted, but our algorithm does not know if it is completed.
The algorithm needs one whole pass without any swap to know it is sorted.
Third Pass:
( 1 2 4 5 8 ) →( 1 2 4 5 8 )
( 1 2 4 5 8 ) →( 1 2 4 5 8 )
( 1 2 4 5 8 ) →( 1 2 4 5 8 )
( 1 2 4 5 8 ) →( 1 2 4 5 8 )

**Bubble Sort Algorithm**

```
1. Repeat Step 2 and 3 for k=1 to n
2. Set ptr=1
3.    Repeat while ptr<n-k
a)    If (A[ptr] > A[ptr+1]) Then
Interchange A[ptr] and A[ptr+1]
[End of If]
b)    ptr=ptr+1
[end of step 3 loop]
[end of step 1 loop]
4.    E
xit
```

Above is the algorithm, to sort an array using Bubble Sort. Although the above logic will sort and unsorted array, still the above algorithm isn't efficient and can be enhanced further. Because as per the above logic, the for loop will keep going for six iterations even if the array gets sorted after the second iteration.

Hence we can insert a flag and can keep checking whether swapping of elements is taking place or not. If no swapping is taking place that means the array is sorted and we can jump out of the for loop.

```
int a[6] = {5, 1, 6, 2, 4, 3};
int i, j, temp;
for(i=0; i<6, i++)
```

```
{
for(j=0; j<6-i-1; j++)
{
int flag = 0;          //taking a flag variable
if( a[j] > a[j+1])
{
temp = a[j];
a[j] = a[j+1];
a[j+1] = temp;
flag = 1;              //setting flag as 1, if swapping occurs
}
}
if(!flag)              //breaking out of for loop if no swapping takes place
{break;}}
```

In the above code, if in a complete single cycle of j iteration(inner for loop), no swapping takes place, and flag remains 0, then we will break out of the for loops, because the array has already been sorted.

## Topic: Character arrays and strings

**Strings, Common Functions in String**
**FUNDAMENTALS OF STRINGS**

- A string is a series of characters treated as a single unit. A string may include letters, digits and various special characters such as +, -, *, / and $. String literals or string constants in C are written in double quotation marks as follows:

|  |  |
|---|---|
| "1000 Main Street" | (a street address) |
| "(080)329-7082" | (a telephone number) |
| "Kalamazoo, New York" | (a city) |

- In C language strings are stored in array of char type along with null terminating character '\0' at the end.
- When sizing the string array we need to add plus one to the actual size of the string to make space for the null terminating character, '\0'.

syntax:
char fname[4];
The above statement declares a string called fname that can take up to 3 characters. It can be indexed just as a regular array as well.
fname[]={'t','w','o'};

Generalized syntax is:-char str[size];
when we declare the string in this way, we can store size-1 characters in the array because the last character would be the null character. For example,
char mesg[10]; can store maximum of 9 characters.
If we want to print a string from a variable, such as four name string above we can do this.
e.g., printf("First name:%s",fname);

We can insert more than one variable. Conversion specification %s is used to insert a string and then go to each %s in our string, we are printing.
A string is an array of characters. Hence it can be indexed like an array.
char ourstr[6] = "EED";

– ourstr[0] is 'E'
– ourstr[1] is 'E'
– ourstr[2] is 'D'
– ourstr[3] is '\0'
– ourstr[4] is '\0'
– ourstr[5] is '\0'

**Reading strings:**
If we declare a string by writing
char str[100];
then str can be read from the user by using three ways;

1. Using scanf() function
2. Using gets() function

3. Using getchar(), getch(), or getche() function repeatedly

The string can be read using scanf() by writing scanf("%s",str);

Although the syntax of scanf() function is well known and easy to use, the main pitfall with this function is that it terminates as soon as it finds a blank space. For example, if the user enters Hello World, then str will contain only Hello. This is because the moment a blank space is encountered, the string is terminated by the scanf() function.Example:

```
char str[10];
printf("Enter a string\n"); scanf("%s",str);
```

The next method of reading a string a string is by using gets() function. The string can be read by writing

```
gets(str);
```

gets() is a function that overcomes the drawbacks of scanf(). The gets() function takes the starting address of the string which will hold the input. The string inputted using gets() is automatically terminated with a null character.

Example:

```
char str[10];
printf("Enter a string\n"); gets(str);
```

The string can also be read by calling the getchar() repeatedly to read a sequence of single characters (unless a terminating character is encountered) and simultaneously storing it in a character array as follows:

```
int i=0;
char str[10],ch;
getchar(ch);
while(ch!='\0')
{
str[i]=ch;                    // store the read character in str
i++;
getch(ch);                   // get another character
}
str[i]='\0';                  // terminate str with null character
```

**Writing string**

The string can be displayed on screen using three ways:

1. Using printf() function
2. Using puts() function
3. Using putchar() function repeatedly

The string can be displayed using pintf() by writing
printf("%s",str);

We can use width and precision specification along with %s. The width specifies the minimum output field width and the precision specifies the maximum number of characters to be displayed. Example:

```
printf("%5.3s",str);
```

this statement would print only the first three characters in a total field of five charaters; also these three characters are right justified in the allocated width.

The next method of writing a string is by using the puts() function. The string can be displayed by writing:

```
puts(str);
```

It terminates the line with a newline character ('\n'). It returns an EOF(-1) if an error occurs and returns a positive number on success.

Finally the string can be written by calling the putchar( ) function repeatedly to print a sequence of single characters.

```
int i=0;
char str[10];
while(str[i]!='\0')
{putchar(str[i]);// print the character on the screen
i++;
}
```

Example: Read and display a string

```
#include<stdio.h>
#include<conio.h> void main()
{char str[20]; clrscr();
printf("\n Enter a string:\n"); gets(str);
scanf("The string is:\n"); puts(str);
getch(); }
```

**COMMON FUNCTIONS IN STRING**

| Type | Method | Description |
|---|---|---|
| char | strcpy(s1, s2) | Copy string |
| char | strcat(s1, s2) | Append string |
| int | strcmp(s1, s2) | Compare 2 strings |
| int | strlen(s) | Return string length |
| char | strchr(s, int c) | Find a character in string |
| char | strstr(s1, s2) | Find string s2 in string s1 |

**strcpy():**
It is used to copy one string to another string. The content of the second string is copied to the content of the first string.
Syntax:
strcpy (string 1, string 2);
Example:
char mystr[10];
mystr = “Hello”; // Error! Illegal !!! Because we are assigning the value to mystr which is not possible in case of an string. We can only use "=" at declarations of C-String.
strcpy(mystr, “Hello”);
It sets value of mystr equal to “Hello”.

**strcmp():**
It is used to compare the contents of the two strings. If any mismatch occurs then it results the difference of ASCII values between the first occurrence of 2 different characters.
Syntax:
int strcmp(string 1, string 2);
Example:
char mystr_a[10] = “Hello”; char mystr_b[10] = “Goodbye”;
– mystr_a == mystr_b; // NOT allowed! The correct way is
if (strcmp(mystr_a, mystr_b ))
printf ("Strings are NOT the same."); else
printf( "Strings are the same.");
Here it will check the ASCII value of H and G i.e, 72 and 71 and return the diference 1.

**strcat():**
It is used to concatenate i.e, combine the content of two strings.
Syntax:
strcat(string 1, string 2);
Example:
char fname[30]={“bob”};
char lname[]={“by”};
printf(“%s”, strcat(fname,lname));
Output:
bobby.
**strlen():**
It is used to return the length of a string.
Syntax:
int strlen(string);

Example:
char fname[30]={“bob”};
int length=strlen(fname);
It will return 3
**strchr():**
It is used to find a character in the string and returns the index of occurrence of the character for the first time in the string.
Syntax:
strchr(cstr);

Example:
char mystr[] = "This is a simple string";
char pch = strchr(mystr,‘s’);

The output of pch is mystr[3]
**strstr():**
It is used to return the existence of one string inside another string and it results the starting index of the string.
Syntax:
strstr(cstr1, cstr2);
Example:
Char mystr[]="This is a simple string";
char pch = strstr(mystr, “simple”);
here pch will point to mystr[10]
**String input/output library functions**

| Function prototype | Function description |
|---|---|
| int getchar(void); | Inputs the next character from the standard input and returns it as integer |
| int putchar(int c); | Prints the character stored in c and returns it as an integer |
| int puts( char s); | Prints the string s followed by new line character. Returns a non-zero integer if possible or EOF if an error occurs |
| int sprint(char s, char format,….) | Equivalent to printf,except the output is stored in the array s instead of printed in the screen. Returns the no.of characters written to s, or EOF if an error occurs |
| int sprint(char s, char format,….) | Equivalent to scanf, except the input is read from the array s rather than from the keyboard. Returns the no.of items successfully read by the function , or EOF if an error occurs |

**Topic: Structure, Use of pointers in self-referential structures Structure Array of structures**

**Definition**

- A Structure is a user defined data type that can store related information together. The variable within a structure are of different data types and each has a name that is used to select it from the structure. C arrays allow you to define type of variables that can hold several data items of the same kind but **structure** is another user defined data type available in C programming, which allows you to combine data items of different kinds.
- Structures are used to represent a record, Suppose you want to keep track of your books in a library. You might want to track the following attributes about each book:
  - Title
  - Author
  - Subject
  - Book ID

- **Structure Declaration-**It is declared using a keyword struct followed by the name of the structure. The variables of the structure are declared within the structure.

Example:

```
Struct struct-name
{
data_type var-name;
data_type var-name;
};
```

- **Structure Initialization-**Assigning constants to the members of the structure is called initializing of structure. Syntax:

```
struct struct_name
{
data _type member_name1;
data _type member_name2;
} struct_var={constant1,constant2};
```

- **Accessing the Members of a structure-**A structure member variable is generally accessed using a '.' operator.

Syntax: strcut_var. member_name;

The dot operator is used to select a particular member of the structure. To assign value to the individual Data members of the structure variable stud, we write, stud.roll=01;

stud.name="Rahul";

To input values for data members of the structure variable stud, can be written as, scanf("%d",&stud.roll);

scanf(''%s",&stud.name);

To print the values of structure variable stud, can be written as: printf("%s",stud.roll);

printf("%f',stud.name);

**QUESTIONS**

1. Write a program using structures to read and display the information about an employee.
2. Write a program to read, display, add and subtract two complex numbers.
3. Write a program to enter two points and then calculate the distance between them.

**Passing Structures through pointers**

- Pointer to a structure is a variable that holds the address of a structure. The syntax to declare pointer to a structure can be given as:

strcut struct_name *ptr;
To assign address of stud to the pointer using address operator(&) we would write
ptr_stud=&stud;
To access the members of the structure (->) operator is used.
for example
Ptr_stud->name=Raj;

**SELF REFERENTIAL STRUCTURE**
Self –referential structures are those structures that contain a reference to data of its same type as that of structure.
Example
struct node
{
int val;
struct node*next;
**};**

**Pointers to Structures**

- You can define pointers to structures in very similar way as you define pointer to any other variable as follows:

struct books *struct_pointer;
Now, you can store the address of a structure variable in the above defined pointer variable. To find the address of a structure variable, place the & operator before the structure's name as follows:
struct_pointer = &book1;
To access the members of a structure using a pointer to that structure, you must use the ->
operator as follows:
struct_pointer->title;

1 .Write a program to display, add and subtract two time defined using hour, minutes and values of seconds.
2. Write a program, using pointer to structure, to initialize the members in the structure. Use functions to print the students information.
3. Write a program using an array of pointers to a structure to read and display the data of a student.

## Topic: Union, enumeration

**UNION**

- Union is a collection of variables of different data types, in case of union information can only be stored In one field at any one time.
- A **union** is a special data type available in C that enables you to store different data types in the same memory location.
- You can define a union with many members, but only one member can contain a value at any given time. Unions provide an efficient way of using the same memory location for multi-purpose.

**Declaring Union**

```
union union-name
{
data_type var-name;
data_type var-name;
};
```

- The **union tag** is optional and each member definition is a normal variable definition, such as int i; or float f; or any other valid variable definition. At the end of the union's definition, before the final semicolon, you can specify one or more union variables but it is optional.

- Here is the way you would define a union type named Data which has the three members i, f, and str. Now, a variable of **Data** type can store an integer, a floating-point number, or a string of characters. This means that a single variable ie. same memory location can be used to store multiple types of data. You can use any built-in or user defined data types inside a union based on your requirement.
- The memory occupied by a union will be large enough to hold the largest member of the union. For example, in above example Data type will occupy 20 bytes of memory space because this is the maximum space which can be occupied by character string. Following is the example which will display total memory size occupied by the above union:

**Accessing a Member of a Union**

```
#include <stdio.h>
#include <string.h>
union Data
{int i;
float f;
char str[20];
};
int main( )
{
union Data data;
data.i = 10;
data.f = 220.5;
strcpy( data.str, "C Programming");
printf( "data.i : %d\n", data.i);
printf( "data.f : %f\n", data.f);
printf( "data.str : %s\n", data.str);
return 0;
}
```

**Exercises:**

1. Write a program to define a union and a structure both having exactly the same members. Using the sizeof operator, print the size of structure variable as well as union variable and comment on the result.

2. Write a program to define a structure for a hotel that has the member's mane, address, grade, number of rooms, and room charges. Write a function to print the names of the hotels in a particular grade. Also write a function to print names of a hotel that have room charges less than the specified value.

**Enumerated Types**

- Enumerations are integer types that you define in a program. The definition of an enumeration begins with the keyword enum, possibly followed by an identifier for the enumeration, and contains a list of the type's possible values, with a name for each value:
  enum [*identifier*] { *enumerator-list* };
  The following example defines the enumerated type enum color:
  enum color { black, red, green, yellow, blue, white=7, gray };
  The identifier color is the *tag* of this enumeration. The identifiers in the listblack, red, and so onare the *enumeration constants* , and have the type int. You can use these constants anywhere within their scopeas case constants in a switch statement, for example.
- Each enumeration constant of a given enumerated type represents a certain value, which is determined either implicitly by its position in the list, or explicitly by initialization with a constant expression. A constant without an initialization has the value 0 if it is the first constant in the list, or the value of the preceding constant plus one. Thus in the previous example, the constants listed have the values 0, 1, 2, 3, 4, 7, 8.
  Within an enumerated type's scope, you can use the type in declarations:

```
enum color bgColor = blue,        // Define two variables
        fgColor = yellow;      // of type enum color.
void setFgColor( enum color fgc ); // Declare a function with a parameter
                                   // of type enum color.
```

- An enumerated type always corresponds to one of the standard integer types. Thus your C programs may perform ordinary arithmetic operations with variables of enumerated types. The compiler may select the appropriate integer type depending on the defined values of the enumeration constants. In the previous example, the type char would be sufficient to represent all the values of the enumerated type enum color.Different constants in an enumeration may have the same value:
  enum { OFF, ON, STOP = 0, GO = 1, CLOSED = 0, OPEN = 1 };
  As the preceding example also illustrates, the definition of an enumerated type does not necessarily have to include a tag. Omitting the tag makes sense if you want only to define constants, and not declare any variables of the given type. Defining integer constants in this way is generally preferable to using a long list of #define directives, as the enumeration provides the compiler with the names of the constants as well as their numeric values.

## Topic: Notion of order of complexity

**Complexity**

Complexity of an algorithm is a measure of the amount of time and/or space required by an algorithm for an input of a given size (n).

**What effects run time of an algorithm?**

(a) Computer used, the hardware platform
(b) Representation of abstract data types (ADT's)
(c) Efficiency of compiler
(d) Competence of implementer (programming skills)
(e) Complexity of underlying algorithm
(f) size of the input

We will show that of those above (e) and (f) are generally the most significant

**Time for an algorithm to run t(n)**

A function of input. However, we will attempt to characterize this by the size of the input. We will try and estimate the WORST CASE, and sometimes the BEST CASE, and very rarely the AVERAGE CASE.

**What do we measure?**

In analyzing an algorithm, rather than a piece of code, we will try and predict the number of times "the principle activity" of that algorithm is performed. For example, if we are analyzing a sorting algorithm we might count the number of comparisons performed, and if it is an algorithm to find some optimal solution, the number of times it evaluates a solution. If it is a graph coloring algorithm we might count the number of times we check that a colored node is compatible with its neighbors.

**Worst Case**

is the maximum run time, over all inputs of size n, ignoring effects (a) through (d) above. That is, we only consider the "number of times the principle activity of that algorithm is performed".

**Best Case**

In this case we look at specific instances of input of size n. For example, we might get best behaviour from a sorting algorithm if the input to it is already sorted.

**Average Case**

Arguably, average case is the most useful measure. It might be the case that worst case behaviour is pathological and extremely rare, and that we are more concerned about how the algorithm runs in the general case. Unfortunately this is typically a very difficult thing to measure. Firstly, we must in some way be able to define by what we mean as the "average input of size n". We would need to know a great deal about the distribution of cases throughout all data sets of size n. alternatively we might make a possibly dangerous assumption that all data sets of size n are equally likely. Generally, in order to get a feel for the average case we must resort to an empirical study of the algorithm, and in some way classify the input (and it is only recently with the advent of high performance, low cost computation, that we can seriously consider this option).

**Varied meanings of complexity**

In several scientific fields, "complexity" has a precise meaning:

- **In computational complexity theory**, the amounts of resources required for the execution of algorithms is studied. The most popular type of computational complexity are:

Time complexity:

```
       - How much time it takes to compute
         - Measured by a function T(N)
          N  = Size of the input
      T(N) = Time complexity function
  Order of magnitude: How rapidly T(N) grows when N grows
```

```
  For example: O(N)  O(log N)  O(N²)  O(2N)
```

Space complexity:

```
    - How much memory it takes to compute
```

```
      - Measured by a function S(N).
```

**Applications of Complexity**

Computational complexity theory is the study of the complexity of problems—that is, the difficulty of solving them. Problems can be classified by complexity class according to the time it takes for an algorithm—usually a computer program—to solve them as a function of the problem size. Some problems are difficult to solve, while others are easy.

For example, some difficult problems need algorithms that take an exponential amount of time in terms of the size of the problem to solve. Take the traveling salesman problem, it can be solved in time O(n^2 2^n) (where n is the size of the network to visit—let's say the number of cities the traveling salesman must visit exactly once). As the size of the network of cities grows, the time needed to find the route grows (more than) exponentially.

Even though a problem may be computationally solvable in principle, in actual practice it may not be that simple. These problems might require large amounts of time or an inordinate amount of space. Computational complexity may be approached from many different aspects.

Computational complexity can be investigated on the basis of time, memory or other resources used to solve the problem. Time and space are two of the most important and popular considerations when problems of complexity are analyzed.

## Module – 5 :( Pointer& File Handling)

## Topic: Pointers- Introduction, declaration, applications

### An Introduction to Pointers

#### Pointer Notation

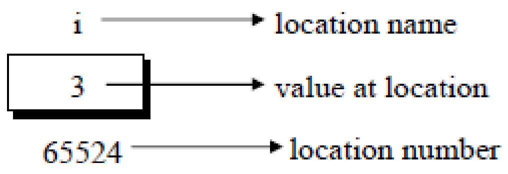
Consider the declaration,

int i = 3 ;

This declaration tells the C compiler to:

(a) Reserve space in memory to hold the integer value.
(b) Associate the name **i** with this memory location.
(c) Store the value 3 at this location.

We may represent **i**'s location in memory by the following memory map.



- We see that the computer has selected memory location 65524 as the place to store the value 3. The location number 65524 is not a number to be relied upon, because some other time the computer may choose a different location for storing the value 3. The important point is, **i**'s address in memory is a number.
- We can print this address number through the following program:

```
main( )
{
int i = 3 ;
printf ( "\nAddress of i = %u", &i ) ;
printf ( "\nValue of i = %d", i ) ;
}
```

The output of the above program would be:

```
Address of i = 65524
Value of i = 3
```

- Look at the first **printf( )** statement carefully. '&' used in this statement is C's 'address of' operator. The expression **&i** returns the address of the variable **i**, which in this case happens to be 65524. Since 65524 represents an address, there is no question of a sign being associated with it. Hence it is printed out using **%u**, which is a format specifier for printing an unsigned integer.
- We have been using the '&' operator all the time in the **scanf( )** statement. The other pointer operator available in C is '*', called 'value at address' operator. It gives the value stored at a particular address. The 'value at address' operator is also called 'indirection' operator.

Observe carefully the output of the following program:

```
main( )
{
int i = 3 ;
printf ( "\nAddress of i = %u", &i ) ;
printf ( "\nValue of i = %d", i ) ;
printf ( "\nValue of i = %d", *( &i ) ) ;
}
```

The output of the above program would be:

```
Address of i = 65524
Value of i = 3
Value of i = 3
```

- Note that printing the value of ***( &i )** is same as printing the value of **i**. The expression **&i** gives the address of the variable **i**. This address can be collected in a variable, by saying, j = &i ;

Here is a program another program.

```
main( )
{
int i = 3 ;
int *j ; j = &i ;
printf ( "\nAddress of i = %u", &i ) ;
printf ( "\nAddress of i = %u", j ) ;
printf ( "\nAddress of j = %u", &j ) ;
printf ( "\nValue of j = %u", j ) ;
printf ( "\nValue of i = %d", i ) ;
printf ( "\nValue of i = %d", *( &i ) ) ;
printf ( "\nValue of i = %d", *j ) ;
}
```

The output of the above program would be: Address of i = 65524

```
Address of i = 65524
Address of j = 65522
Value of j = 65524
Value of i = 3
Value of i = 3
Value of i = 3
```

Look at the following declarations,

```
int *alpha ;
char *ch ;
float *s ;
```

- Here, **alpha, ch** and **s** are declared as pointer variables, i.e. variables capable of holding addresses. Remember that, addresses (location nos.) are always going to be whole numbers, therefore pointers always contain whole numbers. Now we can put these two facts together and say—pointers are variables that contain addresses, and since addresses are always whole numbers, pointers would always contain whole numbers.
- The declaration **float *s** does not mean that **s** is going to contain a floating-point value. What it means is, **s** is going to contain the address of a floating-point value. Similarly, **char *ch** means that **ch** is going to contain the address of a char value. Or in other words, the value at address stored in **ch** is going to be a **char**.

**Pointers**

Consider the following example:

```
main( )
{ int i = 3, *x ;
 float j = 1.5, *y ;
 char k = 'c', *z ;
printf ( "\nValue of i = %d", i ) ;
 printf ( "\nValue of j = %f", j ) ;
 printf ( "\nValue of k = %c", k ) ;
 x = &i ;
y = &j ;
z = &k ;
 printf ( "\nOriginal address in x = %u", x ) ;
printf ( "\nOriginal address in y = %u", y ) ;
 printf ( "\nOriginal address in z = %u", z ) ;
x++ ;
y++ ;
 z++ ;
printf ( "\nNew address in x = %u", x ) ;
 printf ( "\nNew address in y = %u", y ) ;
 printf ( "\nNew address in z = %u", z ) ;
}
```

Here is the output of the program.

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

- Observe the last three lines of the output. 65526 is original value in **x** plus 2, 65524 is original value in **y** plus 4, and 65520 is original value in **z** plus 1. This so happens because every time a pointer is incremented it points to the immediately next location of its type.
- That is why, when the integer pointer **x** is incremented, it points to an address two locations after the current location, since an **int** is always 2 bytes long (under Windows/Linux since **int** is 4 bytes long, new value of **x** would be 65528).
- Similarly, **y** points to an address 4 locations after the current location and **z** points 1 location after the current location. This is a very important result and can be effectively used while passing the entire array to a function.
- The way a pointer can be incremented, it can be decremented as well, to point to earlier locations. Thus, the following operations can be performed on a pointer:

(a) Addition of a number to a pointer. For example,

```
int i = 4, *j, *k ; j = &i ; j = j + 1 ; j = j + 9 ; k = j + 3 ;
```

(b) Subtraction of a number from a pointer. For example,

```
j = &i ; j = j - 2 ; j = j - 5 ; k = j - 6 ;
```

(c) Subtraction of one pointer from another.

One pointer variable can be subtracted from another provided both variables point to elements of the same array. The resulting value indicates the number of bytes separating the corresponding array elements. This is illustrated in the following program.

```
main( ) {
 int arr[ ] = { 10, 20, 30, 45, 67, 56, 74 } ;
 int *i, *j ;
i = &arr[1] ;
      j = &arr[5] ;
printf ( "%d %d", j - i, *j - *i ) ;
```

}

Here **i** and **j** have been declared as integer pointers holding addresses of first and fifth element of the array respectively.

Suppose the array begins at location 65502, then the elements **arr[1]** and **arr[5]** would be present at locations 65504 and 65512 respectively, since each integer in the array occupies two bytes in memory. The expression **j - i** would print a value 4 and not 8. This is because **j** and **i** are pointing to locations that are 4 integers apart. What would be the result of the expression ***j - *i**? 36, since ***j** and ***i** return the values present at addresses contained in the pointers **j** and **i**.

(d) Comparison of two pointer variables

Pointer variables can be compared provided both variables point to objects of the same data type. Such comparisons can be useful when both pointer variables point to elements of the same array. The comparison can test for either equality or inequality. Moreover, a pointer variable can be compared with zero (usually expressed as NULL). The following program illustrates how the comparison is carried out.

```
main( ) { int arr[ ] = { 10, 20, 36, 72, 45, 36 } ; int *j, *k ;
j = &arr [ 4 ] ; k = ( arr + 4 ) ;
if ( j == k ) printf ( "The two pointers point to the same location" ) ; else printf ( "The two pointers do not point to the same location" ) ; }
```

- A word of caution! Do not attempt the following operations on pointers... they would never work out.

(a) Addition of two pointers

(b) Multiplication of a pointer with a constant

(c) Division of a pointer with a constant

Now we will try to correlate the following two facts, which we have learnt above:

(a) Array elements are always stored in contiguous memory locations.

(b) A pointer when incremented always points to an immediately next location of its type.

**Topic: Introduction to dynamic memory allocation (malloc, calloc, realloc, free)**

Dynamic memory management refers to manual memory management. This allows you to obtain more memory when required and release it when not necessary.

Although C inherently does not have any technique to allocate memory dynamically, there are 4 library functions defined under <stdlib.h> for dynamic memory allocation.

| Function | Use of Function |
|---|---|
| malloc() | Allocates requested size of bytes and returns a pointer first byte of allocated space |
| calloc() | Allocates space for an array elements, initializes to zero and then returns a pointer to memory |
| free() | deallocate the previously allocated space |
| realloc() | Change the size of previously allocated space |

**malloc()**

The name malloc stands for "memory allocation".
The function malloc() reserves a block of memory of specified size and return a pointer of type void which can be casted into pointer of any form.

**Syntax of malloc()**

```
ptr = (cast-type*) malloc(byte-size)
```

Here, ptr is pointer of cast-type. The malloc() function returns a pointer to an area of memory with size of byte size. If the space is insufficient, allocation fails and returns NULL pointer.

```
ptr = (int*) malloc(100 * sizeof(int));
```

This statement will allocate either 200 or 400 according to size of int 2 or 4 bytes respectively and the pointer points to the address of first byte of memory.

**calloc()**

The name calloc stands for "contiguous allocation".
The only difference between malloc() and calloc() is that, malloc() allocates single block of memory whereas calloc() allocates multiple blocks of memory each of same size and sets all bytes to zero.

**Syntax of calloc()**

```
ptr = (cast-type*)calloc(n, element-size);
```

This statement will allocate contiguous space in memory for an array of n elements. For example:

```
ptr = (float*) calloc(25, sizeof(float));
```

This statement allocates contiguous space in memory for an array of 25 elements each of size of float, i.e, 4 bytes.

**free()**

Dynamically allocated memory created with either calloc() or malloc() doesn't get freed on its own. You must explicitly use free() to release the space.

**syntax of free()**

```
free(ptr);
```

This statement frees the space allocated in the memory pointed by ptr.

**Example #1: Using C malloc() and free()**

**Write a C program to find sum of n elements entered by user. To perform this program, allocate memory dynamically using malloc() function.**

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

int main()
{
   int num, i, *ptr, sum = 0;
   printf("Enter number of elements: ");
```

```
    scanf("%d", &num);
    ptr = (int*) malloc(num * sizeof(int));  //memory allocated using malloc
    if(ptr == NULL)
    {
        printf("Error! memory not allocated.");
        exit(0);
    }
    printf("Enter elements of array: ");
    for(i = 0; i < num; ++i)
    {
        scanf("%d", ptr + i);
        sum += *(ptr + i);
    }

    printf("Sum = %d", sum);
    free(ptr);
    return 0;
}
```

**Example #2: Using C calloc() and free()**

**Write a C program to find sum of n elements entered by user. To perform this program, allocate memory dynamically using calloc() function.**

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

int main()
{
    int num, i, *ptr, sum = 0;
    printf("Enter number of elements: ");
    scanf("%d", &num);

    ptr = (int*) calloc(num, sizeof(int));
    if(ptr == NULL)
    {
        printf("Error! memory not allocated.");
        exit(0);
    }

    printf("Enter elements of array: ");
    for(i = 0; i < num; ++i)
    {
        scanf("%d", ptr + i);
        sum += *(ptr + i);
    }

    printf("Sum = %d", sum);
    free(ptr);
    return 0;
}
```

**C realloc()**

If the previously allocated memory is insufficient or more than required, you can change the previously allocated memory size using realloc().

**Syntax of realloc()**

```
ptr = realloc(ptr, newsize);
```

Here, ptr is reallocated with size of newsize.

**Example #3: Using realloc()**

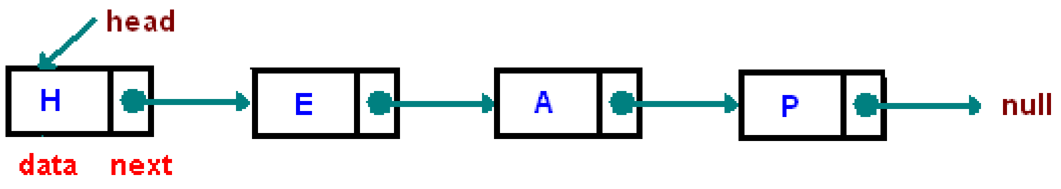
```
#include <stdio.h>
#include <stdlib.h>

int main()
{
   int *ptr, i , n1, n2;
   printf("Enter size of array: ");
   scanf("%d", &n1);

   ptr = (int*) malloc(n1 * sizeof(int));

   printf("Address of previously allocated memory: ");
   for(i = 0; i < n1; ++i)
        printf("%u\t",ptr + i);

   printf("\nEnter new size of array: ");
   scanf("%d", &n2);
   ptr = realloc(ptr, n2 * sizeof(int));
   for(i = 0; i < n2; ++i)
        printf("%u\t", ptr + i);
   return 0;
}
```

## Topic: Notion of linked list (no implementation)

### Introduction

One disadvantage of using arrays to store data is that arrays are static structures and therefore cannot be easily extended or reduced to fit the data set. Arrays are also expensive to maintain new insertions and deletions. In this chapter we consider another data structure called Linked Lists that addresses some of the limitations of arrays.

A linked list is a linear data structure where each element is a separate object.



Each element (we will call it a **node**) of a list is comprising of two items - the data and a reference to the next node. The last node has a reference to null. The entry point into a linked list is called the **head** of the list. It should be noted that head is not a separate node, but the reference to the first node. If the list is empty then the head is a null reference.

A linked list is a dynamic data structure. The number of nodes in a list is not fixed and can grow and shrink on demand. Any application which has to deal with an unknown number of objects will need to use a linked list.

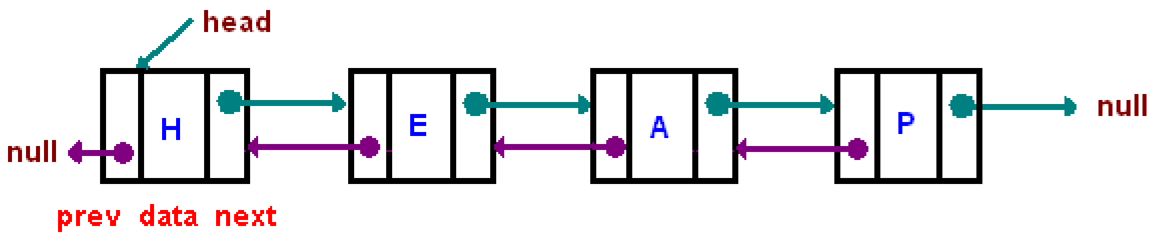
One disadvantage of a linked list against an array is that it does not allow direct access to the individual elements. If you want to access a particular item then you have to start at the head and follow the references until you get to that item.

Another disadvantage is that a linked list uses more memory compare with an array - we extra 4 bytes (on 32-bit CPU) to store a reference to the next node.

### Types of Linked Lists

A **singly linked list** is described above

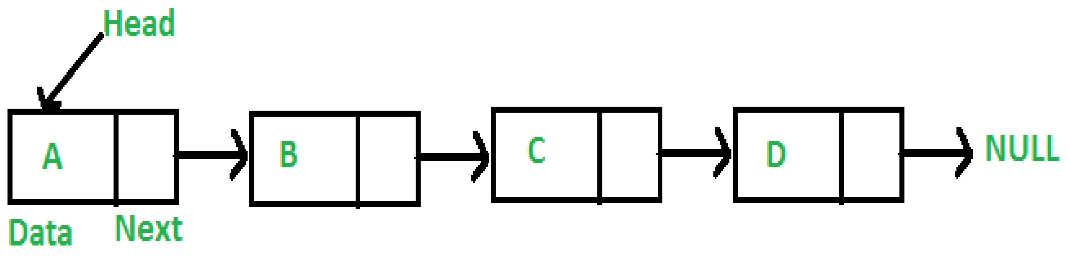
A **doubly linked list** is a list that has two references, one to the next node and another to previous node.



Another important type of a linked list is called a **circular linked list** where last node of the list points back to the first node (or the head) of the list.

### Applications of linked list data structure

A linked list is a linear data structure, in which the elements are not stored at contiguous memory locations. The elements in a linked list are linked using pointers as shown in the below image:



**Applications of linked list in computer science** –

1. Implementation of stacks and queues
2. Implementation of graphs : Adjacency list representation of graphs is most popular which is uses linked list to store adjacent vertices.
3. Dynamic memory allocation: We use linked list of free blocks.
4. Maintaining directory of names
5. Performing arithmetic operations on long integers
6. Manipulation of polynomials by storing constants in the node of linked list
7. representing sparse matrices

**Applications of linked list in real world-**

1. *Image viewer* – Previous and next images are linked, hence can be accessed by next and previous button.
2. *Previous and next page in web browser* – We can access previous and next url searched in web browser by pressing back and next button since, they are linked as linked list.
3. *Music Player* – Songs in music player are linked to previous and next song. you can play songs either from starting or ending of the list.

## Topic: File handling: File I/O functions

**FILE**

- A File is a collection of data stored on a secondary storage device like hard disk. File operation is to combine all the input data into a file and then to operate through the C program. Various operations like insertion, deletion, opening closing etc can be done upon a file.

- When the program is terminated, the entire data is lost in C programming. If you want to keep large volume of data, it is time consuming to enter the entire data. But, if file is created these information can be accessed using few commands.
- There are large numbers of functions to handle file I/O in C language. In this tutorial, you will learn to handle standard I/O(High level file I/O functions) in C. High level file I/O functions can be categorized as:
  1. Text file
  2. Binary file

- A file can be open in several modes for these operations. The various modes are:

**r** open a text file for reading
**w** truncate to zero length or create a text file for writing
**a** append; open or create text file for writing at end-of-file
**rb** open binary file for reading
**wb** truncate to zero length or create a binary file for writing **ab** append; open or create binary file for writing at end-of-file **r+** open text file for update (reading and writing)
**w+** truncate to zero length or create a text file for update **a+** append; open or create text file for update
**r+b or rb+** open binary file for update (reading and writing)
**w+b or wb+** truncate to zero length or create a binary file for update **a+b or ab+** append; open or create binary file for update

- fopen and freopen opens the file whose name is in the string pointed to by filename and associates a stream with it. Both return a pointer to the object controlling the stream, or if the open operation fails a null pointer.
- The error and end-of-file(EOF) indicators are cleared, and if the open operation fails error is set. freopen differs from fopen in that the file pointed to by stream is closed first when already open and any close errors are ignored.

**Q1. Write a program to open a file using fopen().**
**Ans:**

```
#include<stdio.h> void main()
{
fopen() file *fp;
fp=fopen("student.DAT", "r");
if(fp==NULL)
{
printf("The file could not be open"); exit(0);
}
```

**Q2. Write a C program to read name and marks of n number of students from user and store them in a file. If the file previously exits, add the information of n students.**

Ans:

```
#include <stdio.h>
int main()
{
char name[50]; int marks, i,n;
printf("Enter number of students");
scanf("%d", &n);
FILE *fptr; fptr=(fopen("C:\\student.txt","a"));
if (fptr==NULL){ printf("Error!"); exit(1);
}
for(i=0;i<n;++i)
{ printf("For student%d\nEnter name: ",i+1); scanf("%s",name);

printf("Enter marks");
scanf("%d", &marks);
fprintf(fptr, "\nName: %s\nMarks=%d\n", name, marks);
}
 fclose(fptr);
Return 0;
}
```

The fclose function causes the stream pointed to be flushed and the associated file to be closed. Any unwritten buffered data for the stream are delivered to the host environment to be written to the file; any unread buffered data are discarded. The stream is disassociated from the file. If the associated buffer was automatically allocated, it is deallocated. The function returns zero if the stream was successfully closed or **EOF** if any errors were detected.

**Q.3. Write a program to read data from file and close using fclose function.**

**Ans:**

```
#include <stdio.h>
int main()
int n
FILE *fptr;
if ((fptr=fopen("C:\\program.txt","r"))==NULL){
printf("Error! opening file");
exit(1);                    // Program exits if file pointer returns NULL.
}
fscanf(fptr,"%d",&n);
printf("Value of n=%d",n);
fclose(fptr);
return 0;
}
```

**Q4. Write a C program to write all the members of an array of strcures to a file using fwrite(). Read the array from the file and display on the screen.**

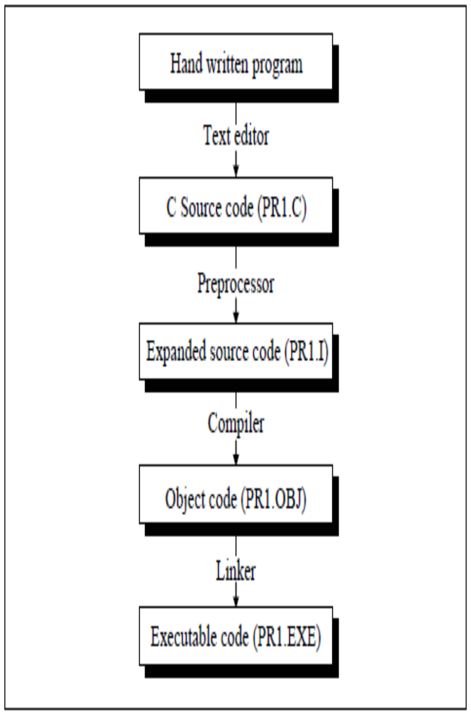
Ans:

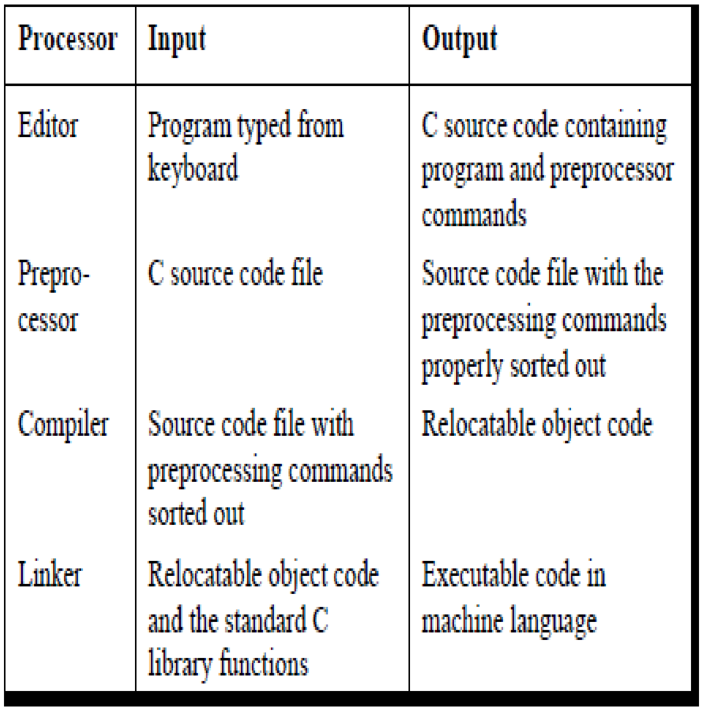
```
#include<stdio.h>
Struct s
{
Char name[50];
Int height;
};
Int main()
{
Struct s a[5], b[5];
FILE *fptr;
Int I;
Fptr=fopen("file.txt", "wb");
For(i=0; i<5; ++i)
{
fflush(stdin);
printf("Enter name: ") ;
gets(a[i].name);
printf("Enter height: ");
scanf("%d",&a[i].height);
}
fwrite(a,sizeof(a),1,fptr);
fclose(fptr);
fptr=fopen("file.txt","rb");
fread(b,sizeof(b),1,fptr);
for(i=0;i<5;++i)
{
printf("Name: %s\nHeight: %d",b[i].name,b[i].height);
}
fclose(fptr);
}
```

**Topic: Standard C preprocessors, defining and calling macros, command-line arguments Conditional compilation, passing values to the compiler**

**Features of C Preprocessor**

- There are several steps involved from the stage of writing a C program to the stage of getting it executed.
- Note that if the source code is stored in a file PR1.C then the expanded source code gets stored in a file PR1.I. When this expanded source code is compiled the object code gets stored in PR1.OBJ. When this object code is linked with the object code of library functions the resultant executable code gets stored in PR1.EXE.
- The preprocessor offers several features called preprocessor directives. Each of these preprocessor directives begin with a # symbol. The directives can be placed anywhere in a program but are most often placed at the beginning of a program, before the first function definition. We would learn the following preprocessor directives here:





| Processor | Input | Output |
|---|---|---|
| Editor | Program typed from keyboard | C source code containing program and preprocessor commands |
| Prepro-cessor | C source code file | Source code file with the preprocessing commands properly sorted out |
| Compiler | Source code file with preprocessing commands sorted out | Relocatable object code |
| Linker | Relocatable object code and the standard C library functions | Executable code in machine language |

- (a) Macro expansion b) File inclusion
  (c) Conditional Compilation
  (d) Miscellaneous directives
  Let us understand these features of preprocessor one by one.

**a) Macro Expansion**

Have a look at the following program.

```
#define UPPER 25
main( )
{
int i ;
for ( i = 1 ; i <= UPPER ; i++ )
printf ( "\n%d", i ) ;
}
```

In this program instead of writing 25 in the **for** loop we are writing it in the form of UPPER, which has already been defined before **main( )** through the statement,

```
#define UPPER 25
```

- This statement is called 'macro definition' or more commonly, just a 'macro'. What purpose does it serve? During preprocessing, the preprocessor replaces every occurrence of UPPER in the program with 25.
- When we compile the program, before the source code passes to the compiler it is examined by the C preprocessor for any macro definitions. When it sees the **#define** directive, it goes through the entire program in search of the macro templates; wherever it finds one, it replaces the macro template with the appropriate macro expansion. Only after this procedure has been completed is the program handed over to the compiler.
- Remember that a macro definition is never to be terminated by a semicolon.

**Macros with Arguments**

The macros that we have used so far are called simple macros. Macros can have arguments, just as functions can. Here is an example that illustrates this fact.

```
#define AREA(x) ( 3.14 * x * x )
main( )
{
float r1 = 6.25, r2 = 2.5, a ;
a = AREA ( r1 ) ;
printf ( "\nArea of circle = %f", a ) ;
a = AREA ( r2 ) ;
printf ( "\nArea of circle = %f", a ) ;
}
```

Here's the output of the program...

```
Area of circle = 122.656250
Area of circle = 19.625000
```

In this program wherever the preprocessor finds the phrase **AREA(x)** it expands it into the statement **( 3.14 * x * x )**. However, that's not all that it does. The **x** in the macro template **AREA(x)** is an argument that matches the **x** in the macro expansion **( 3.14 * x * x )**. The statement **AREA(r1)** in the program causes the variable **r1** to be substituted for **x**. Thus the statement **AREA(r1)** is equivalent to:

- Be careful not to leave a blank between the macro template and its argument while defining the macro. For example, there should be no blank between **AREA** and **(x)** in the definition, #define AREA(x) ( 3.14 * x * x )

**Macros versus Functions**

- when is it best to use macros with arguments and when is it better to use a function? Usually macros make the program run faster but increase the program size, whereas functions make the program smaller and compact.
- If we use a macro hundred times in a program, the macro expansion goes into our source code at hundred different places, thus increasing the program size. On the other hand, if a

function is used, then even if it is called from hundred different places in the program, it would take the same amount of space in the program.

- But passing arguments to a function and getting back the returned value does take time and would therefore slow down the program. This gets avoided with macros since they have already been expanded and placed in the source code before compilation.
- Moral of the story is—if the macro is simple and sweet like in our examples, it makes nice shorthand and avoids the overheads associated with function calls. On the other hand, if we have a fairly large macro and it is used fairly often, perhaps we ought to replace it with a function.

**b) File Inclusion**

The second preprocessor directive we'll explore in this chapter is file inclusion. This directive causes one file to be included in another. The preprocessor command for file inclusion looks like this:

```
#include "filename"
```

and it simply causes the entire contents of **filename** to be inserted into the source code at that point in the program.

**c) Conditional Compilation**

We can, if we want, have the compiler skip over part of a source code by inserting the preprocessing commands **#ifdef** and **#endif**, which have the general form:

```
#ifdef macroname
statement 1 ;
statement 2 ;
statement 3 ;
#endif
```

If **macroname** has been **#define**d, the block of code will be processed as usual; otherwise not.
Where would **#ifdef** be useful? When would you like to compile only a part of your program? In three cases:

**(a)** To "comment out" obsolete lines of code. It often happens that a program is changed at the last minute to satisfy a client. This involves rewriting some part of source code to the client's satisfaction and deleting the old code. But veteran programmers are familiar with the clients who change their mind and want the old code back again just the way it was. Now you would definitely not like to retype the deleted code again.
One solution in such a situation is to put the old code within a pair of /* */ combination. But we might have already written a comment in the code that we are about to "comment out". This would mean we end up with nested comments. Obviously, this solution won't work since we can't nest comments in C.
Therefore the solution is to use conditional compilation as shown below.

```
main( )
{
#ifdef OKAY
statement 1 ;
statement 2 ; /* detects virus */
statement 3 ;
statement 4 ; /* specific to stone virus */
#endif
statement 5 ;
statement 6 ;
statement 7 ;
}
```

Here, statements 1, 2, 3 and 4 would get compiled only if the macro OKAY has been defined, and we have purposefully omitted the definition of the macro OKAY. At a later date, if we want that these statements should also get compiled all that we are required to do is to delete the **#ifdef** and **#endif** statements.

**(b)** A more sophisticated use of **#ifdef** has to do with making the programs portable, i.e. to make them work on two totally different computers. Suppose an organization has two different types of computers and you are expected to write a program that works on both the machines. You can do so by isolating the lines of code that must be different for each machine by marking them off with **#ifdef**. For example:

```
main( )
{
#ifdef INTEL
code suitable for a Intel PC
#else
code suitable for a Motorola PC
#endif
code common to both the computers
}
```

When you compile this program it would compile only the code suitable for a Intel PC and the common code. This is because the macro INTEL has not been defined. Note that the working of **#ifdef - #else - #endif** is similar to the ordinary **if - else** control instruction of C. If you want to run your program on a Motorola PC, just add a statement at the top saying,

```
#define INTEL
```

Sometimes, instead of **#ifdef** the **#ifndef** directive is used. The **#ifndef** (which means 'if not defined') works exactly opposite to **#ifdef**. The above example if written using **#ifndef**, would look like this:

```
main( )
{
#ifndef INTEL
code suitable for a Intel PC
#else
code suitable for a Motorola PC
#endif
code common to both the computers
}
```

**(c)** Suppose a function **myfunc( )** is defined in a file 'myfile.h' which is **#include**d in a file 'myfile1.h'. Now in your program file if you **#include** both 'myfile.h' and 'myfile1.h' the compiler flashes an error 'Multiple declaration for **myfunc'**. This is because the same file 'myfile.h' gets included twice. To avoid this we can write following code in the header file.

```
/* myfile.h */
#ifndef __myfile_h
#define __myfile_h
myfunc( )
{
/* some code */
}
#endif
```

First time the file 'myfile.h' gets included the preprocessor checks whether a macro called **__myfile_h** has been defined or not. If it has not been then it gets defined and the rest of the code gets included. Next time we attempt to include the same file, the inclusion is prevented since **__myfile_h** already stands defined. Note that there is nothing special about **__myfile_h**. In its place we can use any other macro as well.

## ***#if* and *#elif* Directives**

The **#if** directive can be used to test whether an expression evaluates to a nonzero value or not. If the result of the expression is nonzero, then subsequent lines upto a **#else**, **#elif** or **#endif** are compiled, otherwise they are skipped. A simple example of **#if** directive is shown below:

```
main( )
{
#if TEST <= 5
statement 1 ;
statement 2 ;
statement 3 ;
#else
statement 4 ;
statement 5 ;
statement 6 ;
#endif
}
```

If the expression, **TEST <= 5** evaluates to true then statements 1, 2 and 3 are compiled otherwise statements 4, 5 and 6 are compiled. In place of the expression **TEST <= 5** other expressions like **( LEVEL == HIGH || LEVEL == LOW )** or **ADAPTER == CGA** can also be used.

If we so desire we can have nested conditional compilation directives. An example that uses such directives is shown below.

```
#if ADAPTER == VGA
code for video graphics array
#else
#if ADAPTER == SVGA
code for super video graphics array
#else
code for extended graphics adapter
#endif
#endif
```

The above program segment can be made more compact by using another conditional compilation directive called **#elif**. The same program using this directive can be rewritten as shown below. Observe that by using the **#elif** directives the number of #**endif**s used in the program get reduced.

```
#if ADAPTER == VGA
code for video graphics array
#elif ADAPTER == SVGA
code for super video graphics array
#else
code for extended graphics adapter
#endif
```

**d) Miscellaneous Directives**

There are two more preprocessor directives available, though they are not very commonly used. They are:

(a) #undef

(b) #pragma

***#undef* Directive**

On some occasions it may be desirable to cause a defined name to become 'undefined'. This can be accomplished by means of the **#undef** directive. In order to undefine a macro that has been earlier **#define**d, the directive,

#undef macro template

can be used. Thus the statement,

#undef PENTIUM

would cause the definition of PENTIUM to be removed from the system. All subsequent **#ifdef PENTIUM** statements would evaluate to false. In practice seldom are you required to undefine a macro, but for some reason if you are required to, then you know that there is something to fall back upon.

***#pragma* Directive**

This directive is another special-purpose directive that you can use to turn on or off certain features. Pragmas vary from one compiler to another. There are certain pragmas available with Microsoft C compiler that deal with formatting source listings and placing comments in the object file generated by the compiler. Turbo C/C++ compiler has got a pragma that allows you to suppress warnings generated by the compiler. Some of these pragmas are discussed below.

(a) **#pragma startup** and **#pragma exit:** These directives allow us to specify functions that are called upon program startup (before **main( )**) or program exit (just before the program terminates). Their usage is as follows:

```
void fun1( ) ;
void fun2( ) ;
#pragma startup fun1
#pragma exit fun2
main( )
{
printf ( "\nInside maim" ) ;
}
void fun1( )
{
printf ( "\nInside fun1" ) ;
}
void fun2( )
{
printf ( "\nInside fun2" ) ;
}
```

And here is the output of the program.

```
Inside fun1
Inside main
Inside fun2
```

Note that the functions **fun1( )** and **fun2( )** should neither receive nor return any value. If we want two functions to get executed at startup then their pragmas should be defined in the reverse order in which you want to get them called.

(b) **#pragma warn:** This directive tells the compiler whether or not we want to suppress a specific warning. Usage of this pragma is shown below.

```
#pragma warn –rvl /* return value */
#pragma warn –par /* parameter not used */
#pragma warn –rch /* unreachable code */
int f1( )
{
int a = 5 ;
}
void f2 ( int x )
{
printf ( "\nInside f2" ) ;
}
int f3( )
{
int x = 6 ;
return x ;
x++ ;
}
void main( )

{
f1( ) ;
f2 ( 7 ) ;
f3( ) ;
}
```

If you go through the program you can notice three problems immediately. These are:
(a) Though promised, **f1( )** doesn't return a value.
(b) The parameter **x** that is passed to **f2( )** is not being used anywhere in **f2( )**.
(c) The control can never reach **x++** in **f3( )**.
If we compile the program we should expect warnings indicating the above problems. However, this does not happen since we have suppressed the warnings using the **#pragma** directives. If we replace the '–' sign with a '+' then these warnings would be flashed on compilation. Though it is a bad practice to suppress warnings, at times it becomes useful to suppress them. For example, if you have written a huge program and are trying to compile it, then to begin with you are more interested in locating the errors, rather than the warnings. At such times you may suppress the warnings. Once you have located all errors, then you may turn on the warnings and sort them out.

## Important Programs with solution

### 1. Program to find area and circumference of circle.

```
#include<stdio.h>
#include<conio.h>
void main()
{
int r;
float pi=3.14,area,ci;
clrscr();
printf("enter radius of circle: ");
scanf("%d",&r);
area=pi*r*r;
printf("area of circle=%f ",area);
ci=2*pi*r;
printf("circumference=%f ",ci);
getch();
}
```

Output:

```
enter radius of a circle: 5
area of circle=78.000
circumference=31.4
```

### 2. Program to find the simple interest.

```
#include<stdio.h>
#include<conio.h>
void main()
{
int p,r,t,si;
clrscr();
printf("enter principle, Rate of interest & time to find simple interest: ");
scanf("%d%d%d",&p,&r,&t);
si=(p*r*t)/100;
printf("simple intrest= %d",si);
getch();
}
```

Output:

```
enter principle, rate of interest & time to find simple interest: 500
5
2
simple interest=50
```

## 3. Program to convert temperature from degree centigrade to Fahrenheit.

```
#include<stdio.h>
#include<conio.h>
void main()
{
float c,f;
clrscr();
printf("enter temp in centigrade: ");
scanf("%f",&c);
f=(1.8*c)+32;
printf("temp in Fahrenheit=%f ",f);
getch();
}
```

Output:

```
enter temp in centigrade: 32
temp in Fahrenheit=89.59998
```

## 4. Program to calculate sum of 5 subjects and find percentage.

```
#include<stdio.h>
#include<conio.h>
void main()
{
int s1,s2,s3,s4,s5,sum,total=500;
float per;
clrscr();
printf("enter marks of 5 subjects: ");
scanf("%d%d%d%d%d",&s1,&s2,&s3,&s4,&s5);
sum=s1+s2+s3+s4+s5;
printf("sum=%d",sum);
per=(sum*100)/total;
printf("percentage=%f",per);
getch();
}
```

Output:

```
enter marks of 5 subjects: 60
65
50
60
60
sum=300
percentage=60.000
```

**5. Program to find maximum between three numbers using nested if**

```
#include <stdio.h>
int main()
{    int num1, num2, num3, maximum;
     printf("Enter three numbers to find maximum: \n");
    scanf("%d%d%d", &num1, &num2, &num3);
     if(num1>num2)
    {       if(num1>num3)
        {           maximum = num1;
        }
        else
        {           maximum = num3;
        }
    }
    else
    {       if(num2>num3)
        {           maximum = num2;
        }
        else
        {           maximum = num3;
        }
    }
     printf("\nMaximum among all three numbers = %d", maximum);
     return 0;
}
```

**6. C program check even or odd using if else**

```
#include <stdio.h>
int main()
{    int num;
     printf("Enter any number to check even or odd: ");
    scanf("%d", &num);
     if(num%2 == 0)
    {       printf("Number is Even.\n");
    }
    else
    {       printf("Number is Odd.\n");
    }
     return 0;
}
```

## 7. C program to check Leap Year

It is a special year containing one extra day i.e. total 366 days in ayear. If the year is exactly divisible by 4 and not divisible by 100 then its Leap Year Else if the year is exactly divisible by 400 then its Leap Year Else its a common year.

```
#include <stdio.h>
int main()
{   int year;
    printf("Enter year : ");
    scanf("%d", &year);
    if(((year%4 == 0) && (year%100 !=0)) || (year%400==0))
    {       printf("LEAP YEAR");
    }
    else
    {       printf("COMMON YEAR");
    }
    return 0;
}
```

## 8. Program to print day of week

```
#include <stdio.h>
int main()
{   int week;
     printf("Enter week number (1-7): ");
    scanf("%d", &week);
     if(week == 1)
    {       printf("MONDAY\n");
    }
    else if(week == 2)
    {       printf("TUESDAY\n");
    }
    else if(week == 3)
    {       printf("WEDNESDAY\n");
    }
    else if(week == 4)
    {       printf("THURSDAY\n");
    }
    else if(week == 5)
    {       printf("FRIDAY\n");
    }
    else if(week == 6)
    {       printf("SATURDAY\n");
    }
    else if(week == 7)
    {       printf("SUNDAY\n");
    }
    else
    {       printf("Invalid Input! Please enter week number between 1-7.\n");
    }
     return 0;
}
```

## 9. C program to find all roots of a quadratic equation

```
    #include <stdio.h>
#include <math.h> //Used for sqrt()
int main()
{   float a, b, c;
  float root1, root2, imaginary;
  float discriminant;
    printf("Enter values of a, b, c of quadratic equation (aX^2 + bX + c): ");
  scanf("%f%f%f", &a, &b, &c);
  discriminant = (b*b) - (4*a*c);
  if(discriminant > 0)
  {       root1 = (-b + sqrt(discriminant)) / (2*a);
    root2 = (-b - sqrt(discriminant)) / (2*a);
    printf("Two distinct and real roots exists: %.2f and %.2f\n", root1, root2);
  }
  else if(discriminant == 0)
  {       root1 = root2 = -b / (2*a);
    printf("Two equal and real roots exists: %.2f and %.2f\n", root1, root2);
  }
  else if(discriminant < 0)
  {       root1 = root2 = -b / (2*a);
    imaginary = sqrt(-discriminant) / (2*a);
    printf("Two distinct complex roots exists: %.2f + i%.2f and %.2f - i%.2f\n", root1,
imaginary, root2, imaginary);
  }
  return 0;
}
```

**10. C program to create calculator using switch case and functions**

```
#include <stdio.h>
int main()
{    char op;
    float num1, num2, result=0;
     printf("WELCOME TO SIMPLE CALCULATOR\n");
    printf("----------------------------\n");
    printf("Enter [number 1] [+ - * /] [number 2]\n");
     scanf("%f %c %f", &num1, &op, &num2);
     switch(op)
    {        case '+': result = num1 + num2;
            break;
        case '-': result = num1 - num2;
            break;
        case '*': result = num1 * num2;
            break;
        case '/': result = num1 / num2;
            break;
        default: printf("Invalid operator");
            return 0;
    }
     printf("%.2f %c %.2f = %.2f\n", num1, op, num2, result);
     return 0;
}
```

**11. C program to count number of digits in an integer**

Method to count number of digits in an integer.

1. Initialize a variable count to 0.
2. Increment count by 1 if num > 0 (where num is the number whose digits count is to be found).
3. Remove the last digit from the number as it isn't needed anymore. Hence divide the number by 10 i.e num = num / 10.
4. If number is greater than 0 then repeat steps 2-4.

```
#include <stdio.h>
 int main()
{   long long num;
    int count = 0;
     printf("Enter any number: ");
    scanf("%lld", &num);
     while(num != 0)
    {       count++;
       num /= 10; // num = num / 10
    }
     printf("Total digits: %d", count);
     return 0;
}
```

**12. C program to find sum of digits of a number**

logic of this program can be divided in three basic steps:

1. Find the last digit of number by performing modular division.
2. Add the last digit just found above to sum.
3. Remove the last digit from number by dividing the number by 10.

Repeat the above three steps till the number becomes 0 and you will be left with the sum of digits.

```
        #include <stdio.h>
int main()
{   int num, sum=0;
  printf("Enter any number to find sum of its digit: ");
   scanf("%d", &num);
       while(num!=0)
   {       sum += num % 10;
       num = num / 10;
   }
   printf("\nSum of digits = %d", sum);
   return 0;
}
```

**13. C program to calculate product of digits of a number**

logic of this program can be divided in three basic steps:

1. Find the last digit of number by performing modular division.
2. Multiply the last digit just found above to product.
3. Remove the last digit from number by dividing the number by 10.

Repeat the above three steps till the number becomes 0 and you will be left with the product of digits.

```
      #include <stdio.h>
int main()
{    int n;
    long product=1;
     printf("Enter any number to calculate product of digit: ");
    scanf("%d", &n);
     while(n!=0)
    {       product = product * (n % 10);
        n = n / 10;
    }
    printf("\nProduct of digits = %ld", product);
     return 0;
}
```

**14. C program to find reverse of any number**

The process of reversing involves four basic steps:

1. Multiply the rev variable by 10.
2. Find the last digit of the given number.
3. Add the last digit just found to rev.
4. Divide the original number by 10 to eliminate the last digit, which is not needed anymore.

Repeat the above four steps till the original number becomes 0 and finally we will be left with the reversed number in rev variable.

```
      #include <stdio.h>
 int main()
{    int n, rev = 0;
     /* Reads the number from user */
    printf("Enter any number to find reverse: ");
    scanf("%d", &n);
     /* Repeats the steps till n becomes 0 */
    while(n!=0)
    {
       /* Multiples rev by 10 and adds the last digit to it*/
       rev = (rev * 10) + (n % 10);
        /* Eliminates the last digit from num */
       n = n/10;
    }
     printf("Reverse = %d", rev);
     return 0;
}
```

**15. C program to check whether a number is palindrome or not**

After retrieving the reverse of number we just need to make a conditional check that whether the given number is equal to its reverse or not. If the given number and its reverse are same then the number is palindrome otherwise not.

```
      #include <stdio.h>
 int main()
{    int n, num, rev = 0;
```

```
     /* Reads a number from user */
    printf("Enter any number to check palindrome: ");
    scanf("%d", &n);
     num = n; //Copies original value to num.
     /* Finds reverse of n and stores in rev */
    while(n!=0)
    {       rev = (rev * 10) + (n % 10);
        n = n/10;
    }
     /* Check if reverse is equal to original num or not */
    if(rev==num)
    {       printf("%d is palindrome.", num);
    }
    else
    {       printf("%d is not palindrome.", num);
    }
     return 0;
}
```

**16. C program to find factorial of any number**

Example:

Input number: 5
Output factorial: 120

Logic to find factorial

Finding factorial is really easy you just need to know how to iterate over a loop. Finding factorial can be basically divided two steps:

1. Run a loop from 1 to n (where n is the number whose factorial is to be found).
2. Multiply the current loop counter value with fact (where fact is a variable that stores factorial and is initially initialized with 1).

```
#include <stdio.h>
 int main()
{
    int i, num;
    long long fact=1;
     /* Reads a number from user */
    printf("Enter any number to calculate factorial: ");
    scanf("%d", &num);
     /* Runs a loop from 1 to n */
    for(i=1; i<=num; i++)
    {
        fact = fact * i;
    }
    printf("Factorial of %d = %lld", num, fact);
     return 0;
}
```

**17. C program to check whether a number is prime number or not**

Example:

Enter any number: 17
Output: Number is Prime number
Prime numbers

Prime numbers are the positive integers greater than 1 that is only divisible by 1 and self. For example: 2, 3, 5, 7, 11 etc...

```
#include <stdio.h>
 int main()
{
    int i, n, flag;
     //Flag is used as notification. Initially we have supposed that the number is prime.
    flag = 1;

    /* Reads a number from user */
    printf("Enter any number to check prime: ");
    scanf("%d", &n);

    for(i=2; i<=n/2; i++)
    {
        /*
         * If the number is divisible by any number
         * other than 1 and self then it is not prime
         */
        if(n%i==0)
        {
            flag = 0;
            break;
        }
    }

    /*
     * If flag contains 1 then it is prime
     */
    if(flag==1)
    {
        printf("\n%d is prime number", n);
    }
    else
    {
        printf("\n%d is not prime number", n);
    }
    return 0;
}
```

**18. C program to check whether a number is Armstrong number or not**

Example:

Input: 371

Output: Armstrong number

Armstrong number

Armstrong number is a special number whose sum of cube of its digits is equal to the original number. For example: 371 is an Armstrong number because

33 + 73 + 13 = 371

Logic to check armstrong number

Checking of armstrong number can be divided into basic three steps:

1. Find last digit of the given number simply by performing modular division by 10.
2. Find cube of the last digit just found and add it to a variable called sum.
3. Repeat above two steps till the number becomes 0. And at the completion of the iteration check whether sum equals to original number or not. If it does then the given number is armstrong number otherwise not.

```
#include <stdio.h>
 int main()
{
    int n, num, lastDigit, sum = 0;
    /* Reads a number from user */
    printf("Enter any number to check Armstrong number: ");
    scanf("%d", &n);

    /* Copies the original value of n to num */
    num = n;

    /*
     * Finds the sum of cubes of each digit of number
     */
    while(n != 0)
    {
        /* Finds last digit of number */
        lastDigit = n % 10;
                /* Finds cube of last digit and adds to sum */
        sum += (lastDigit * lastDigit * lastDigit);

        n = n / 10;
    }
    /* Checks if sum of cube of digits is equal to original value or not. */
    if(num == sum)
    {
        printf("\n%d is Armstrong number.", num);
    }
    else
    {
        printf("\n%d is not Armstrong number.", num);
    }
     return 0;}
```

**19. C program to check whether a number is perfect number or not**

Example:

Input number: 6

Output: 6 is perfect number

*Perfect number*

A perfect number is a positive integer which is equal to the sum of its proper positive divisors. For example: 6 is the first perfect number

Proper divisors of 6 are 1, 2, 3

And 1+2+3 = 6. Hence 6 is a perfect number

```
#include <stdio.h>
 int main()
{    int i, num, sum = 0;

    /* Reads a number from user */
    printf("Enter any number to check perfect number: ");
    scanf("%d", &num);

    /* Finds the sum of all proper divisors */
    for(i=1; i<num; i++)
    {
        /* If i is a divisor of num */
        if(num%i==0)
        {
            sum += i;
        }
    }

    /* Checks whether the sum of proper divisors is equal to num */
    if(sum == num)
    {
        printf("\n%d is a Perfect number", num);
    }
    else
    {
        printf("\n%d is not a Perfect number", num);
    }
     return 0;
}
```

**20. C program to print all prime numbers between 1 to n**

Example:

Input lower limit: 1
Input upper limit: 10
Output prime numbers between 1-10: 2, 3, 5, 7

Prime number is a positive integer greater than 1 that is only divisible by 1 and self.

Example: 2, 3 , 5, 7, 11 are the first five prime numbers

```
#include <stdio.h>
 int main()
{
    int i, j, n, isPrime; //isPrime is used as flag variable

    /* Reads upper limit to print prime */
    printf("Find prime numbers between 1 to : ");
    scanf("%d", &n);
    printf("\nAll prime numbers between 1 to %d are:\n", n);

    /* Finds all Prime numbers between 1 to n */
    for(i=2; i<=n; i++)
    {
        /* Assume that the current number is Prime */
        isPrime = 1;

        /* Check if the current number i is prime or not */
        for(j=2; j<=i/2; j++)
        {
            /*
             * If i is divisible by any number other than 1 and self
             * then it is not prime number
             */
            if(i%j==0)
            {
                isPrime = 0;
                break;
            }
        }

        /* If the number is prime then print */
        if(isPrime==1)
        {
            printf("%d is Prime number\n", i);
        }
    }
     return 0;
}
```

**21. C program to find sum of all prime numbers between 1 to n**

Example:

Input: n=10

Output: Sum of all prime numbers between 1 to 10 = 2 + 3 + 5 + 7 = 17

```
#include <stdio.h>
 int main()
{
    int i, j, n, isPrime, sum=0;
     /*
     * Reads a number from user
     */
    printf("Find sum of all prime between 1 to : ");
    scanf("%d", &n);

    /*
     * Finds all prime numbers between 1 to n
     */
    for(i=2; i<=n; i++)
    {

        /*
         * Checks if the current number i is Prime or not
         */
        isPrime = 1;
        for(j=2; j<=i/2 ;j++)
        {
            if(i%j==0)
            {
                isPrime = 0;
                break;
            }
        }

        /*
         * If i is Prime then add to sum
         */
        if(isPrime==1)
        {
            sum += i;
        }
    }
    printf("Sum of all prime numbers between 1 to %d = %d", n, sum);
    return 0;
}
```

**22. C program to find fibonacci series upto n terms**

Example:

Input upper limit: 10

Output: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34

*Fibonacci series*

Fibonacci series is a series of numbers where the current number is the sum of previous two terms.

For Example: 0, 1, 1, 2, 3, 5, 8, 13, 21...

Fibonacci series algorithm

If you look to the given series very carefully you will find a specific pattern i.e. the current number is the sum of previous two numbers.

Fibonacci series : 0, 1, 1, 2, 3, 5, 8, 13, 21...

1 is the sum of 0 + 1

2 is the sum of 1 + 1

3 is the sum of 2 + 1

5 is the sum of 3 + 2

and so on....

Lets suppose the first number as a i.e. a = 0

Second number as b i.e. b = 1

And lets suppose c as our current number in Fibonacci series also initialize the value with 0 i.e. c = 0

Now Fibonacci series algorithm is simple and contains only four steps.

Step 1: Print the value of c.

```
Step 2: a = b.
Step 3: b = c.
Step 4: c = a + b.
```

```
#include <stdio.h>
int main()
{
   int a, b, c, i, n;
   printf("Enter value of n to print Fibonacci series : ");
   scanf("%d", &n);
   a = 0;
   b = 1;
   c = 0;

   for(i=1; i<=n; i++)
   {
      printf("%d, ", c);

      a=b;
      b=c;
      c=a+b;
   }

   return 0;}
```

**23. C program to check prime, Armstrong, perfect number using functions**

Example:

```
Input any number: 11
Output: 11 is prime number
11 is not a armstrong number
11 is not a perfect number
```

```
    #include <stdio.h>
 /* Function declarations */
int isPrime(int num);
int isArmstrong(int num);
int isPerfect(int num);


int main()
{
   int num;

   printf("Enter any number: ");
   scanf("%d", &num);

   //Calls the isPrime() functions
   if(isPrime(num))
   {
       printf("%d is Prime number.\n", num);
   }
   else
   {
       printf("%d is not Prime number.\n", num);
   }

   //Calls the isArmstrong() function
   if(isArmstrong(num))
   {
       printf("%d is Armstrong number.\n", num);
   }
   else
   {
       printf("%d is not Armstrong number.\n", num);
   }

   //Calls the isPerfect() function
   if(isPerfect(num))
   {
       printf("%d is Perfect number.\n", num);
   }
   else
   {
       printf("%d is not Perfect number.\n", num);
   }

   return 0;
}


/**
 * Checks whether a number is prime or not. Returns 1 if the number is prime
 * otherwise returns 0.
 */
int isPrime(int num)
{
   int i;

   for(i=2; i<=num/2; i++)
   {
       /*
        * If the number is divisible by any number
        * other than 1 and self then it is not prime
        */
       if(num%i == 0)
       {
           return 0;
       }
   }
```

```
        return 1;
}

 /**
 * Checks whether a number is Armstrong number or not. Returns 1 if the number
 * is Armstrong number otherwise returns 0.
 */
int isArmstrong(int num)
{
    int lastDigit, sum, n;
    sum = 0;
    n = num;

    while(n!=0)
    {
        /* Finds last digit of number */
        lastDigit = n % 10;

        /* Finds cube of last digit and adds to sum */
        sum += lastDigit * lastDigit * lastDigit;

        n = n/10;
    }

    return (num == sum);
}

/**
 * Checks whether the number is perfect number or not. Returns 1 if the number
 * is perfect otherwise returns 0.
 */
int isPerfect(int num)
{
    int i, sum, n;
    sum = 0;
    n = num;

    for(i=1; i<n; i++)
    {
        /* If i is a divisor of num */
        if(n%i == 0)
        {
            sum += i;
        }
    }
        return (num == sum);
}
```

## 24. C program to find reverse of a number using recursion

Example:

Input number: 12345
Output reverse: 54321

Logic to find reverse of number using recursion
basic steps i.e.

1. Multiply the rev variable by 10.
2. Find the last digit of the given number.
3. Add the last digit just found to rev.
4. Divide the original number by 10 to eliminate the last digit, which is not needed anymore.

And we repeat the above four steps till the number becomes 0 and we are left with the reversed number in rev variable. Here also we will use the above four steps to find the reverse using recursive approach with the given base condition:

reverse(0) = 0 {Base condition}
reverse(n) = (n%10 * pow(10, digits)) + reverse(n/10) {where digit is the total number of digits in number}

```
#include <stdio.h>
#include <math.h>
  /* Fuction declaration */
int reverse(int num);

int main()
{
    int num, rev;

    /* Reads number from user */
    printf("Enter any number: ");
    scanf("%d", &num);

    /*Calls the function to reverse number */
    rev = reverse(num);

    printf("Reverse of %d = %d", num, rev);

    return 0;
}

/**
 * Recursive function to find reverse of any number
 */
int reverse(int num)
{
    int digit;

    //Base condition
    if(num==0)
        return 0;
    //Finds total number of digits
    digit = (int)log10(num);

    return ((num%10 * pow(10, digit)) + reverse(num/10));
}
```

## 25. C program to generate nth fibonacci term using recursion

**Example:**

   Input any number: 10
   Output 10th fibonacci term: 55

Logic to find nth Fibonacci term
base condition of the recursion which is explained below.

```
   fibo(0) = 0 {Base condition}
   fibo(1) = 1 {Base condition}
   fibo(num) = fibo(num-1) + fibo(num+2)
```

```
#include <stdio.h>
 //Function declaration
long long fibo(int num);
 int main()
{
    int num;
    long long fibonacci;

    //Reads number from user to find nth fibonacci term
    printf("Enter any number to find nth fiboacci term: ");
    scanf("%d", &num);

    fibonacci = fibo(num);

    printf("%dth fibonacci term is %lld", num, fibonacci);

    return 0;
}
 /**
 * Recursive function to find nth Fibonacci term
 */
long long fibo(int num)
{
    if(num == 0) //Base condition
        return 0;
    else if(num == 1) //Base condition
        return 1;
    else
        return fibo(num-1) + fibo(num-2); //Recursively calls fibo() to find nth fibonacci term.
}
```

**26. C program to find sum of all elements of an array**

Example:

    Input elements: 10, 20, 30, 40, 50
    Sum of all elements = 150

```
#include <stdio.h>
#define MAX_SIZE 100

int main()
{
   int arr[MAX_SIZE];
   int i, n, sum=0;

   /*
    * Reads size and elements in array from user
    */
   printf("Enter size of the array: ");
   scanf("%d", &n);
   printf("Enter %d elements in the array: ", n);
   for(i=0; i<n; i++)
   {
      scanf("%d", &arr[i]);
   }

   /*
    * Adds each element of array to sum
    */
   for(i=0; i<n; i++)
   {
      sum = sum + arr[i];
   }

   printf("Sum of all elements of array = %d", sum);

   return 0;
}
```

## 27. C program to find maximum and minimum element in array

Example: If the elements of the array are: 10, 50, 12, 16, 2
Maximum = 50
Minimum = 2

Logic to find maximum/minimum in array

```
Step 1: Read element in array.
    Step 2: Let's suppose the first element of array as maximum. Set max=array[0]
    Step 3: Set i=0
    Step 4: If array[i] > max then Set max=array[i]
    Step 5: Increment i by 1. Set i=i+1
    Step 6: Repeat Step 4-5 till i<size (Where size is the size of array).
```

```c
#include <stdio.h>

int main()
{
    int arr[100];
    int i, max, min, size;

    /*
     * Reads size array and elements in the array
     */
    printf("Enter size of the array: ");
    scanf("%d", &size);
    printf("Enter elements in the array: ");
    for(i=0; i<size; i++)
    {
        scanf("%d", &arr[i]);
    }

    /* Supposes the first element as maximum and minimum */
    max = arr[0];
    min = arr[0];

    /*
     * Finds maximum and minimum in all array elements.
     */
    for(i=1; i<size; i++)
    {
        /* If current element of array is greater than max */
        if(arr[i]>max)
        {
            max = arr[i];
        }

        /* If current element of array is smaller than min */
        if(arr[i]<min)
        {
            min = arr[i];
        }
    }

    /*
     * Prints the maximum and minimum element
     */
    printf("Maximum element = %d\n", max);
    printf("Minimum element = %d", min);

    return 0;
}
```

**28. C program to find reverse of an array**

Example:

```
If the elements of the array are: 10, 5, 16, 35, 500
Then its reverse would be: 500, 35, 16, 5, 10
That is if
array[0] = 10
array[1] = 5
array[2] = 16
array[3] = 35
array[4] = 500

Then after reversing array elements should be
array[0] = 500
array[1] = 35
array[2] = 16
array[3] = 5
array[4] = 10
```

Algorithm:

Basic algorithm for reversing any array.

```
Step 1: Take two array say A and B. A will hold the original values and B will hold the
reversed values.
Step 2: Read elements in array A.
Step 3: Set i=size - 1, j=0 (Where size is the size of array).
Step 4: Set B[j] = A[i]
Step 5: Set j = j + 1 and i = i - 1.
Step 6: Repeat Step 4-5 till i>=0.
Step 7: Print the reversed array in B.
```

Program:

```
#include <stdio.h>

int main()
{
    int arr[100], reverse[100];
    int size, i, j;

    /*
     * Reads the size of array and elements in array
     */
    printf("Enter size of the array: ");
    scanf("%d", &size);
    printf("Enter elements in array: ");
    for(i=0; i<size; i++)
    {
        scanf("%d", &arr[i]);
    }

    /*
     * Reverse the array
     */
    j=0;
    for(i=size-1; i>=0; i--)
    {
        reverse[j] = arr[i];
        j++;
    }

    /*
     * Prints the reversed array
     */
    printf("\nReversed array : ");
    for(i=0; i<size; i++)
    {
        printf("%d\t", reverse[i]);
    }

    return 0;
}
```

**29. C program to search an element in the array**

Example: If the elements of array are: 10, 12, 20, 25, 13, 10, 9, 40, 60, 5
Element to search is: 25
Output: Element exists

Algorithm:

```
Step 1: Read elements in array A.
    Step 2: Read element to be searched in num.
    Step 3: Set i=0, flag=0. We have initially supposed that the number doesn't exists in the
    array hence we set flag=0.
    Step 4: If A[i]==num then set flag=1 and print "Element found" and goto Step 6.
    Step 5: Set i=i+1 and repeat Step 4 till i<size (Where size is the size of array).
    Step 6: If the value of flag=0 then print "Element not found".
```

Program:

```
#include <stdio.h>

int main()
{
    int arr[100];
    int size, i, num, flag;
    /*
     * Read size of array and elements in array
     */
    printf("Enter size of array: ");
    scanf("%d", &size);

    printf("Enter elements in array: ");
    for(i=0; i<size; i++)
    {
        scanf("%d", &arr[i]);
    }

    printf("\nEnter the element to search within the array: ");
    scanf("%d", &num);

    /* Supposes that element is not in the array */
    flag = 0;
    for(i=0; i<size; i++)
    {
        /*
         * If element is found in the array
         */
        if(arr[i]==num)
        {
            flag = 1;
            printf("\n%d is found at position %d", num, i+1);
            break;
        }
    }

    /*
     * If element is not found in array
     */
    if(flag==0)
    {
        printf("\n%d is not found in the array", num);
    }

    return 0;
}
```

### 30. C program to sort an array in ascending order

Example:

If the elements of array are: 20, 2, 10, 6, 52, 31, 0, 45, 79, 40

Array sorted in ascending order: 0, 2, 6, 10, 20, 31, 40, 45, 52, 79

Algorithm:

Here we will use basic algorithm to sort arrays in ascending order:

```
Step 1: Read elements in array.
Step 2: Set i=0
```

```
    Step 3: Set j=i+1
    Step 4: If array[i] > array[j] then swap value of array[i] and array[j].
    Step 5: Set j=j+1
    Step 6: Repeat Step 4-5 till j<n (Where n is the size of the array)
    Step 7: Set i=i+1
    Step 8: Repeat Step 3-7 till i<n
```

Program:

```
#include <stdio.h>

int main()
{
   int arr[100];
   int size, i, j, temp;

   /*
    * Read size of array and elements in array
    */
   printf("Enter size of array: ");
   scanf("%d", &size);

   printf("Enter elements in array: ");
   for(i=0; i<size; i++)
   {
      scanf("%d", &arr[i]);
   }

   /*
    * Array sorting code
    */
   for(i=0; i<size; i++)
   {
      for(j=i+1; j<size; j++)
      {
         /*
          * If there is a smaller element towards right of the array then swap it.
          */
         if(arr[j] < arr[i])
         {
            temp = arr[i];
            arr[i] = arr[j];
            arr[j] = temp;
         }
      }
   }

   /*
    * Prints the sorted array
    */
   printf("\nElements of array in sorted ascending order: ");
   for(i=0; i<size; i++)
```

```
    {
        printf("%d\t", arr[i]);
    }

    return 0;
}
```

### 31. C program to find total number of alphabets, digits or special characters in a string

Example:

    Input string: I love programming.
    Output: Alphabets = 16
    Digits = 0
    Special character = 3
    Total length = 19

```
#include <stdio.h>

#define MAX_SIZE 100 //Maximum size of the string
  int main()
{
    char string[MAX_SIZE];
    int alphabets, digits, others, i;

    alphabets = digits = others = i = 0;


    /* Reads a string from user */
    printf("Enter any string : ");
    gets(string);

    /*
     * Checks each character of string
     */
    while(string[i]!='\0')
    {
        if((string[i]>='a' && string[i]<='z') || (string[i]>='A' && string[i]<='Z'))
        {
            alphabets++;
        }
        else if(string[i]>='0' && string[i]<='9')
        {
            digits++;
        }
        else
        {
            others++;
        }
        i++;
    }

    printf("Alphabets = %d\n", alphabets);
    printf("Digits = %d\n", digits);
    printf("Special characters = %d\n", others);

    return 0;
}
```

- **C program to convert Decimal to Binary number system**

  Example:
  Input any decimal number: 112
  Output binary number: 0111000

*Decimal number system:*

Decimal number system is a base 10 number system. Decimal number system uses only 10 symbols to represent all number i.e. 0 1 2 3 4 5 6 7 8 9

*Binary number system:*

Binary number system is a base 2 number system. Binary number system uses only 2 symbols to represent all numbers i.e. 0 and 1

Algorithm to convert from Decimal to Binary number system:

```
Algorithm Decimal to Binary conversion
begin:
read (DECIMAL);
BINARY ← 0; PLACE ← 1; REM ← 0;
while (DECIMAL !=0) do
begin
   REM ← DECIMAL % 2;
   BINARY ← (REM * PLACE) + BINARY;
   PLACE ← PLACE * 10;
   DECIMAL ← DECIMAL / 2;
end
write('Binary = ' BINARY)
end
```

Program:

```
#include <stdio.h>

int main()
{
    long long decimal, tempDecimal, binary;
    int rem, place = 1;

    binary = 0;

    /*
```

```
     * Reads decimal number from user
     */
    printf("Enter any decimal number: ");
    scanf("%lld", &decimal);
    tempDecimal = decimal;

    /*
     * Converts the decimal number to binary number
     */
    while(tempDecimal!=0)
    {
        rem = tempDecimal % 2;

        binary = (rem * place) + binary;

        tempDecimal /= 2;
        place *= 10;
    }

    printf("\nDecimal number = %lld\n", decimal);
    printf("Binary number = %lld", binary);

    return 0;
}
```

**Questions Bank**

**Section –A (Short answer type questions)**

1. Draw a flow chart for arranging three numbers a, b and c in ascending order.
2. What is the need of an operating system?

**2.** Classify the operating system into different types based on their processing capability.
**3.** Differentiate between a Linker and Compiler.
**4.** What is kernel?
**5.** Explain types of errors while compilation and execution of c program.
**6.** What is top down approach?
**7.** Compare compiler, interpreter and assembler.
**8.** Why ‘C' is called structured programming language?
**9.** What are the classifications of computer? Explain any two in detail.
**10.** Describe the functionalities of an operating system.
**11.** Explain different types of type conversion in 'C' with suitable example.
**12.** What are escape sequences characters?
**13.** What do curly braces denote in C? Why does it make sense to use curly braces to surround the body of a function?
**14.** Give any four format specifiers used in printf() function
**15.** Write down the difference between associatively and precedence of operators.
**16.** If int a = 2 , b = 3, x = 0; Find the value of x = (++a, b+=a).
**17.** Find the value of Y (assume Y is integer data type) Y=4 * 2 / 4 – 6 /2 + 3 % 2 * 6 / 2 + 2 > 2 && 4 != 2.
**18.** Explain the structure of C program with suitable example.
**19.** What is difference between macro and constant?
**20.** What will be the output of the following code?

```
void main( )
{
int a=5,b=6,c;
c=a++>b&&++b<10||b++;
printf("%d%d%d",a,b,c);
}
```

**21.** Find the output of the statement printf("%d%d%d",n++,++n,--n);if n=3.
**22.** Differentiate the following:
1. Unary operator and Binary operator
2. Local variables and Global variables

**23.** Write briefly about the user defined functions and standard library functions.
**24.** Explain the ternary operators in details with example.
**25.** Give various types of scope in ‘C’ programming.
**26.** Write down any four characteristics of a good programming language.

**27.** In C programming what will be the value of r if r = p % q where p = -17 and q= 5?

**28.** Write a program in C to find the total surface area of the cylinder, if the diameter and the height of the cylinder is given.

**29.** Write an algorithm to print the sum of first 100 Fibonacci numbers.

**30.** Write a program in C that takes a year from twentieth century as an input and then tells whether it is a leap year or not ?

**31.** Write a function in C that finds the reverse of a given integer number.

**32.** What do you mean by parameter passing? Discuss various types of parameter passing mechanism in C with example.

**33.** What are Functions? What are advantages of using multiple functions in a program.

**34.** Give the for loop statement to print each of the following sequence of integers:
(i) 1, 2, 4, 8, 16, 32 (ii) -4,-2 ,0 , 2, 4

**35.** What are iterative control statements? Explain the difference between while loop and do-while loop.

**36.** Differentiate the use of break and continue statements. With an example.

**37.** Write a program to check whether a given number is Armstrong number or not.

**38.** Explain the purpose of continue statement in c.

**39.** Explain the output of following:

```
#define SQUARE (x) x * x
void main ( ) {
int i;
i = 8l/SQUARE (9);
printf("%d", i); }
```

**40.** What is function prototype? Why is it required?

**41.** Find the output of the following code :

```
    void main ( ) {
                    switch (0) {
                                    case 0 : printf("0+3");
                                    case 0+1 : printf ("0+5");
                                    default : printf ("Wrong Input"); }}
```

**42.** Draw a flowchart to find the sum and reverse of a given number.

**43.** Write a program in C to convert binary number into decimal numbers.

**44.** Write a program in C to check whether a given number is perfect or not.

**45.** Differentiate between nested If and the switch statement in C language with suitable example.

**46.** Declare and initialize three dimensional array.

**47.** 60. Write a function in C language that returns the sum of the principal diagonal of a two dimensional matrix.

**48.** What are subscripts? How are they specified? What restrictions apply to the values that can be assigned to subscripts in 'C' language?

**49.** Explain ENUM with suitable example.

**50.** What are differences in Array and Structures? Explain with an example.

**51.** Explain UNION with suitable example.

**52.** What is void pointer? How is it different from other pointers?

**53.** What is the task of following memory allocation functions?

1. malloc
2. calloc.

**54.** Explain any two file operations and two bitwise operators with example.

**55.** What do you mean by dynamic memory allocation?

**56.** What is pointer? How many bytes are needed for a pointer ? Justify your answer.

**57.** What is the difference between the following directives? #include<studio.h> and #include "studio.h"

**58.** What is conditional compilation and how does it help a programmer?

**59.** Explain the difference between const char *p and char const *p .

**60.** What do you mean by C Preprocessor? Give example of any two preprocessor directives.

**61.** Write output of the following printf("%d", strcmp("QUIET', "QUILT"));.

**62.** Discuss about #PRAGMA directive.

## Section –B (Long answer type questions)

**1.**How many bytes are needed for a pointer ? Justify your answer .

**2.**Differentiate between Structure and Union.

**3.**Write a function in C to exchange the content of two integer variables without using a third variable.

**4.**Justify that operating system is a resource manager.

**5.** What are the different types of operators in C language? Explain with example. Discuss the significance of each.

**6.** Write an algorithm to print all the even numbers and odd numbers between any given two integers Nl and N2 (N1 < N2) and also print the sum of all even numbers and odd numbers.

**7.** Write short note on the following with example in reference to C language:
(i)Data type (ii) Entry and exit control loops, (iii) Switch statement and if statement.

**8.** Discuss the major components of a digital computer with suitable block diagram. Also discuss the functions of these components.

**9.** Differentiate between the following: -
i) High level language and low level language. (ii) Compiler and interpreter.
(iii) Logical error and run time error. (iv) Algorithm and flowchart.

**10.** Draw a flowchart to play a dice game according to the following rules
(i) If you throw two identical numbers, you win.
(ii) If the numbers are not the same, but are both even, or both odd, you have another turn.
(iii) If one number is odd, and the other is even, you lose.

**11.** For a digital computer briefly explain the following:
a) Cache Memory b) Control Unit c) ALU

**12.** Differentiate between pseudo code and algorithm. Write the characteristics of an algorithm. Draw a flow chart and write a program in C for printing Fibonacci series upto a term given by user.

**13.** Write about the formatted and unformatted Input / Output functions in 'C'

**14.** What are different types in C. Explain in terms of memory size, format specifier and range.

**15.** Write a program in C that accept the length and width of a rectangle and print the area. The area of a rectangle is calculated by a function and returns it value to main program where it is printed. The values of length and width of rectangle are accepted from the keyboard. Also give the flowchart and algorithm.

**16.** Discuss various storage classes in C with suitable example. Also give their significance.

**17.** Draw the flow chart and write a function in c to calculate the sum S of the following series :
$S = 1^1 + 2^2 + 3^3 + 4^4 + ... + N^N$

**18.** What do you mean by sorting? Write a program in C to sort the given n positive integers. Also give the flow chart for the same.

**19.** Write a program that counts the total number of Vowels in a given sentence.

**20.** What is a role of SWITCH statement in C programming language? Give the syntax of SWITCH statement with a suitable example.

**21.** Differentiate between nested If and the switch statement in C language with suitable example.

**22.** Draw a flowchart and write a function in C to calculate factorial of given number and also write a program to calculate the sum of the following series using the above function
s=1! + 2! + 3! +….+N!

**23.** Write a program in C that accepts roll number and name of students of a class size of one hundred students along with the marks obtained by them in Physics, Chemistry and Mathematics. Print roll number and the name of top ten students in the order of merit. The merit is based on the sum of the marks obtained in the three subjects.

**24.** Write a program in C to calculate the sum of the following series upto nt$^{h}$ term
$F(x)=x^1-x^3+x^5-x^7+\ldots$

**25.** Draw flowchart and write a program in C to calculate the sum of Fibonacci series upto 100 terms.

**26.** Write a program in C where user input a 5 digit positive integer and display it as shown in bellow:
Example: Suppose Input integer is 24689 and it is to be displayed as

```
2 4 6 8 9
4 6 8 9
6 8 9
8 9
9
```

**27.** Explain in details all types of loop and conditional statements exist in C .

**28.** What is a recursive function? What are its basic properties? Write a function which finds the sum of first n integers; where n is the argument to the function.

**29.** Write a program to print out our array using the array notation and by dereferencing Pointer.

**30.** Explain recursive function and also write a recursive C program for computing the factorial of a given integer number .

**31.** Write a program in 'C' to convert a number in decimal system which is entered by the user to a number in binary system.

**32.** Describe about the types of looping statements in C' with necessary syntax.

**33.** Write a C program to calculate the sum of the following series upto 50 terms
$SUM = -1^3 + 3^3 - 5^3 + 7^3 - 9^3 +11^3$.

**34.** Write a C program which reverses the digits of the integer input given to it. For example an input 65367 is outputted as 76356.

**35.** Write a program in ‘C’ to print the following

```
A
BA
ABA
BABA
```

ABABA

**36.** In a class there are fifty students enrolled for five subjects. Write a program to enter the marks of different subjects and give average mark of student obtained by him and overall average of the class.

**37.** Write a program in C using Switch Statement to find the value of y for particular value of N. The value of a, x, b, N are to be input through keyboard by user.

If N=1 y=ax%b

If N=2 $y=ax^2 + b^2$

If N=3 y=-bx

If N=4 y=a+x/b

**38.** What is difference in searching and sorting? Write an algorithm to sort a list containing ten numbers.

**39.** Write a program to create an integer array of n elements, pass this array as an argument to a function where it is sorted and displayed.

**40.** Explain with example, the syntax and usage of the following in C program :

(i) Nested Structure Definition (ii) Array of Structures

**41.** A program in 'C' language contains the following declaration :

static int x[8] = {1,2,3,4,5,6,7,81};

(i) What is the meaning of x ? (ii) What is the meaning of (x+2)? ,

(iii) What is the meaning of *x? (iv) What is the meaning of(*x+2)?

(v) What is the meaning of *(x+2)?

**42.** How to declare an array? Explain about various operations of an array.

**43.** Write a 'C' program to read in 10 integers and print their average, minimum and maximum numbers.

**44.** Write a C program to sequentially search a given integer element from a given list of numbers.

**45.** Write a program to construct a 3*3 matrix B that contains only the fractional part of another 3*3 matrix A that contains floating point numbers. Assume that elements of matrix A contain only two digit fractional numbers.

**46.** a)What are the types of function? Write a C program to find the sum of two matrices of size M X N each.Write a program to check whether a square matrix is symmetric or not.

**47.** Write a program in C that reads the two strings of length at least 7, then concatenate these strings.

**48.** Write a program to sort the names of student of a class.

**49.** Draw flow chart and write a program in C to print the multiplication of two matrices A and B of size N X N.

**50.** Write a program in C to read 5X4 matrix using array and to calculate the following:

a) Sum of the elements of third row of matrix b) Sum of all the elements

**51.** Write a program to define a structure named employee having empid, name and designation. Use this structure to store the data in a file named "employee.dat. Once all the records are entered display the employee details stored in a file.

**52.** Define a structure called cricket that will describe the following information :

Player name Team name Batting average

Using cricket declare an array player with 50 elements and write a program to read a information about all the 50 players and print a team-wise list containing the names of players with their batting average.

**53.** What is enumerated data type? Write a C program to display months in the year using enum.

**54.** Differentiate structure and union in 'C'. Write a C program to store the student details using union.

**55.** Write a simple database program in C which stores personal details of 100 persons such as Name, Date of Birth, Address, Phone number etc.

**56.** Declare a structure to which contains the following member and write a program in 'C' to list all students who secured more than 75 marks. ROLL NO.,NAME,FATHER_NAME, AGE,CITY,MARKS

**57.** What do mean by pointers? How pointer variables are initialized? Write a program in C to swap the values of any two integer variables using pointers.

**58.** Write a program in C to copy the text of one file to another.

**59.** What is a pointer? How pointers are declared in C programming language? Illustrate with a suitable example

**60.** Using dynamic memory allocation, write a program in C to Accept element of a matrix of size 3X3 and print transpose of it.

**61.** Write a C program that accepts name and marks scored by students in five subjects. Use functions :

(i) ADD to add records with student names and marks to a file.

(ii) CALCULATE to read the records from the file, calculate the result and display.

**62.** Explain with example defining and calling macros and conditional compilation.

**63.** Write a program in C to determine whether a string given by the user is palindrome or not without using strrev( )function.

**64.** List out various file operations in C. Write a C program to count the number of characters in a file.Write a C program to copy the content of one file into another file.

**65.** A file name DATA contains a series of integer numbers write a program to read these numbers and then write all 'odd' numbers to a file to be called ODD and all even numbers to a file to be called EVEN.

**66.** Write a program in 'C' that takes 10 integers from al file and write square of these integers into another file.

**67.** State the features of pointers. Write a 'C' program to sort a given number using pointers.

**68.** What are the merits and demerits of static and dynamic memory allocation techniques?