LECTURE NOTES

ON

PS&P

B.TECH I YEAR- I SEMESTER

(R19)

DEPARTMENT OF HUMANITIES & SCIENCES

VEMU INSTITUTE OF TECHNOLOGY:: P.KOTHAKOTA Chittoor-Tirupati National Highway, P.Kothakota, Near Pakala, Chittoor (Dt.), AP - 517112 (Approved by AICTE. New Delhi, Affiliated to JNTUA Ananthapuramu and An ISO 9001:2015 Certified Institute)

UNIT-I

Unit 1

Computer Fundamentals: What is a Computer, Evolution of Computers, Generations of Computers, Classification of Computers, Anatomy of a Computer, Memory revisited, Introduction to Operating systems, Operational overview of a CPU.
Introduction to Programming, Algorithms and Flowcharts: Programs and Programming, Programming languages, Compiler, Interpreter, Loader, Linker, Program execution, Fourth generation languages, Fifth generation languages, Classification of Programming languages, Structured programming concept, Algorithms, Pseudo-code, Flowcharts, Strategy for designing algorithms, Tracing an algorithm to depict logic, Specification for converting algorithms into programs.

What is a Computer?

COMPUTER stands for Common Operating Machine Purposely Used for Technological and Educational Research.

A **computer** is a machine that accepts data as input, process that data using programs, and output the processed data and also is used to perform arithmetic and logical operations automatically.

- A computer is a data processor. It can accept an alphanumeric input, which may be either data or instructions or both.
- The computer remembers the input by storing it in memory cells.
- Internally it converts the input data to meaningful binary digits, performs the instructed operations on the binary data, and transforms the data from binary digit form to understandable alphanumeric form.
- It gives out the result of the arithmetic or logical computations as output information.

Characteristics of Computer:

Speed: The computer can process the calculations with at most speed than other.
Storage Capacity: The computer can provide large amount of space to store data, instructions, and Software etc.
Accuracy: The computer can process the calculations with at most accuracy to provide effective results. **Reliability:** The computers are reliable one why because, if any error occurs, those errors can be resolved to continue with execution.
Versatility /Adaptability: The computers can perform various tasks and can be used for different Purposes.
Diligence/Un-tiredness: The computers can perform repetitive calculations any number of times with same level of accuracy

Evolution of Computers

- **Computing in the mechanical era :**
  - The first mechanical calculating apparatus was the **abacus,** which was invented in 500 BC in Babylon.
  - It was used extensively without any improvement until 1642 when Blaise Pascal designed a calculator that employed gears and wheels.

- **Computing in the electrical era :**
    - A mechanical machine, driven by a single electric motor, was developed in 1889 by Herman Hollerith to count, sort, and collate data stored on punched cards.
    - In 1941, Konrad Zuse developed the first electronic calculating computer, **Z3**… and so on

Generations of Computers

The history of computers can be divided into generations, with each generation defined by a technological breakthrough

Generation 0 (1642-1940):

- **Technology** – Mechanical gears, hand-crank, dials and knobs (Gears and Relays)
- **Speed** – Very slowest computing devices
- **Programming Languages** – No specific languages, instead of that patterns used
- **Examples** – Analytical Engine, Difference Engine
- **Disadvantage**s – Very Large, Bulky and noisy.

First Generation (1940-1956):

- **Technology –** Vacuum Tubes
- **Speed –**Slowest computing devices
- **Programming Languages –** Machine level language
- **Examples –** EDSAC ,EDVAC
- **Disadvantages –** Large, Bulky and difficult to program

Second Generation (1956-1963):

- **Technology** -- Transistor
- **Speed** – Faster than first generation
- **Programming Languages** – Assembly level language
- **Examples** – IBM-1401, IBM-1620
- **Disadvantages** – High cost and limited to special purpose tasks

Third Generation (1964-1975):

- **Technology –** Integrated circuits (ICs)
- **Speed –** Faster than second generation
- **Programming Languages –** High level languages
- **Examples –** IBM-360, HoneyWell-6000
- **Disadvantages** -- Limited storage capacity

Fourth Generation (1975-1989):

- **Technology -** VLSI (Very Large Scale Integration Circuits)
- **Speed –** Faster than third generation
- **Programming Languages –** High level languages
- **Examples –** IBM PC series , Apple Series
- **Disadvantages –** Difficult to manufacture

Fifth Generation (1989 – till date):

- **Technology –** ULSI (Ultra Large Scale Integrated Circuits)
- **Speed –** Fastest of all times
- **Programming Languages –** HLL , Integrated Development Environment (IDE)
- **Examples –** Laptop , PDA (Personal Digital assistance)

- **Disadvantages –** Lack of human like Intelligence.

Classification of Computers

The computer systems can be classified into three types ``

1. On the basis of size.
2. On the basis of functionality.
3. On the basis of data handling.

Classification on the basis of size

1. **Super computers:** The super computers are the highest performing system. A supercomputer is a computer with a high level of performance compared to a general-purpose computer. The actual Performance of a supercomputer is measured in FLOPS instead of MIPS. All of the world's fastest 500 supercomputers run Linux-based operating systems. Additional research is being conducted in China, the US, the EU, Taiwan and Japan to build even faster, higher performing and more technologically superior supercomputers. Supercomputers actually play an important role in the field of computation, and are used for intensive computation tasks in various fields, including quantum mechanics, weather forecasting, climate research, oil and gas exploration, molecular modelling, and physical simulations. And also throughout the history, supercomputers have been essential in the field of the cryptanalysis.
   Eg: PARAM, jaguar, roadrunner.

2. **Mainframe computers :** These are commonly called as big iron, they are usually used by big organisations for bulk data processing such as statics, census data processing, transaction processing and are widely used as the severs as these systems has a higher processing capability as compared to the other classes of computers, most of these mainframe architectures were established in 1960s, the research and development worked continuously over the years and the mainframes of today are far more better than the earlier ones, in size, capacity and efficiency.
   Eg: IBM z Series, System z9 and System z10 servers.

3. **Mini computers :** These computers came into the market in mid 1960s and were sold at a much cheaper price than the main frames, they were actually designed for control, instrumentation, human interaction, and communication switching as distinct from calculation and record keeping, later they became very popular for personal uses with evolution.
   In the 60s to describe the smaller computers that became possible with the use of transistors and core memory technologies, minimal instructions sets and less expensive peripherals such as the ubiquitous Teletype Model 33 ASR.They usually took up one or a few inch rack cabinets, compared with the large mainframes that could fill a room, there was a new term "MINICOMPUTERS" coined **Eg:** Personal Laptop, PC etc.

4. **Microcomputers:** A microcomputer is a small, relatively inexpensive computer with a microprocessor as its CPU. It includes a microprocessor, memory, and minimal I/O circuitry mounted on a single printed circuit board.The previous to these computers, mainframes and minicomputers, were comparatively much larger, hard to maintain and more expensive. They actually formed the foundation for present day microcomputers and smart gadgets that we use in day to day life.
   Eg: Tablets, Smart watches.

Classification on the basis of functionality

1. **Servers:** Servers are nothing but dedicated computers which are set-up to offer some services to the clients. They are named depending on the type of service they offered. Eg: security server, database server.
2. **Workstation:** Those are the computers designed to primarily to be used by single user at a time. They run multi-user operating systems. They are the ones which we use for our day to day personal / commercial work.
3. **Information Appliances:** They are the portable devices which are designed to perform a limited set of tasks like basic calculations, playing multimedia, browsing internet etc. They are generally referred as the mobile devices. They have very limited memory and flexibility and generally run on "as-is" basis.
4. **Embedded computers:** They are the computing devices which are used in other machines to serve limited set of requirements. They follow instructions from the non-volatile memory and they are not required to execute reboot or reset. The processing units used in such device work to those basic requirements only and are different from the ones that are used in personal computers- better known as workstations.

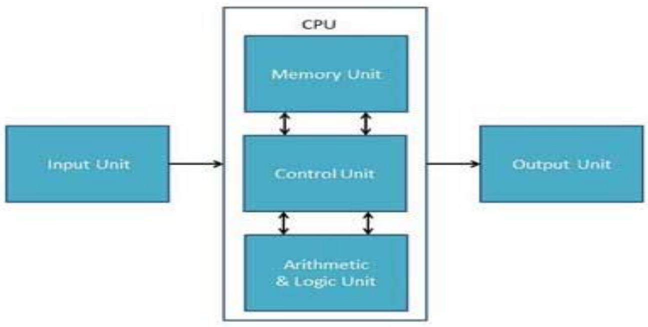
Classification on the basis of data handling

1. **Analog :** An analog computer is a form of computer that uses the continuously-changeable aspects of physical fact such as electrical, mechanical, or hydraulic quantities to model the problem being solved. Anything that is variable with respect to time and continuous can be claimed as analog just like an analog clock measures time by means of the distance travelled for the spokes of the clock around the circular dial.

2. **Digital :** A computer that performs calculations and logical operations with quantities represented as digits, usually in the binary number system of "0" and "1", "Computer capable of solving problems by processing information expressed in discrete form. From manipulation of the combinations of the binary digits, it can perform mathematical calculations, organize and analyze data, control industrial and other processes, and simulate dynamic systems such as global weather patterns.

3. **Hybrid:** A computer that processes both analog and digital data, Hybrid computer is a digital computer that accepts analog signals, converts them to digital and processes them in digital form.

Anatomy of a Computer



The same basic logical structure and perform the following five basic operations for converting raw input data into information useful to their users.

S.No.	Operation	Description
1	Take Input	The process of entering data and instructions into the computer system.
2	Store Data	Saving data and instructions so that they are available for processing as and when required.
3	Processing Data	Performing arithmetic, and logical operations on data in order to convert them into useful information.
4	Output Information	The process of producing useful information or results for the user, such as a printed report or visual display.
5	Control the workflow	Directs the manner and sequence in which all of the above operations are performed.

Input Unit

This unit contains devices with the help of which we enter data into the computer. This unit creates a link between the user and the computer. The input devices translate the information into a form understandable by the computer.

CPU (Central Processing Unit)

CPU is considered as the brain of the computer. CPU performs all types of data processing operations. It stores data, intermediate results, and instructions (program). It controls the operation of all parts of the computer.

- ALU (Arithmetic Logic Unit)
- Memory Unit
- Control Unit

Output Unit

The output unit consists of devices with the help of which we get the information from the computer. This unit is a link between the computer and the users. Output devices translate the computer's output into a form understandable by the users.

INPUT DEVICES

The important input devices which are used in a computer

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

Keyboard

Keyboard is the most common and very popular input device which helps to input data to the computer. The layout of the keyboard is like that of traditional typewriter, although there are some additional keys provided for performing additional functions.

Keyboards are of two sizes 84 keys or 101/102 keys, but now keyboards with 104 keys or 108 keys are also available for Windows and Internet.

The keys on the keyboard are as follows

S.No	Keys & Description
1	**Typing Keys** These keys include the letter keys (A-Z) and digit keys (09) which generally give the same layout as that of typewriters.
2	**Numeric Keypad** It is used to enter the numeric data or cursor movement. Generally, it consists of a set of 17 keys that are laid out in the same configuration used by most adding machines and

	calculators.
3	**Function Keys** The twelve function keys are present on the keyboard which are arranged in a row at the top of the keyboard. Each function key has a unique meaning and is used for some specific purpose.
4	**Control keys** These keys provide cursor and screen control. It includes four directional arrow keys. Control keys also include Home, End, Insert, Delete, Page Up, Page Down, Control(Ctrl), Alternate(Alt), Escape(Esc).
5	**Special Purpose Keys** Keyboard also contains some special purpose keys such as Enter, Shift, Caps Lock, Num Lock, Space bar, Tab, and Print Screen.

Mouse

Mouse is the most popular pointing device. It is a very famous cursor-control device having a small palm size box with a round ball at its base, which senses the movement of the mouse and sends corresponding signals to the CPU when the mouse buttons are pressed.

Generally, it has two buttons called the left and the right button and a wheel is present between the buttons. A mouse can be used to control the position of the cursor on the screen, but it cannot be used to enter text into the computer.

Advantages

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

Joystick

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

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

Light Pen

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

When the tip of a light pen is moved over the monitor screen and the pen button is pressed, its photocell sensing element detects the screen location and sends the corresponding signal to the CPU.

Track Ball

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

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

Scanner

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

Scanner captures images from the source which are then converted into a digital form that can be stored on the disk. These images can be edited before they are printed.

Digitizer

Digitizer is an input device which converts analog information into digital form. Digitizer can convert a signal from the television or camera into a series of numbers that could be stored in a computer. They can be used by the computer to create a picture of whatever the camera had been pointed at.

Digitizer is also known as Tablet or Graphics Tablet as it converts graphics and pictorial data into binary inputs. A graphic tablet as digitizer is used for fine works of drawing and image manipulation applications.

Microphone

Microphone is an input device to input sound that is then stored in a digital form.

The microphone is used for various applications such as adding sound to a multimedia presentation or for mixing music.

Magnetic Ink Card Reader (MICR)

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

This reading process is called Magnetic Ink Character Recognition (MICR). The main advantages of MICR is that it is fast and less error prone.

Optical Character Reader (OCR)

OCR is an input device used to read a printed text.

OCR scans the text optically, character by character, converts them into a machine readable code, and stores the text on the system memory.

Bar Code Readers

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

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

Optical Mark Reader (OMR)

OMR is a special type of optical scanner used to recognize the type of mark made by pen or pencil. It is used where one out of a few alternatives is to be selected and marked.

It is specially used for checking the answer sheets of examinations having multiple choice questions.

Output devices

The important output devices used in a computer.

- Monitors
- Graphic Plotter
- Printer

Monitors

Monitors, commonly called as **Visual Display Unit** (VDU), are the main output device of a computer. It forms images from tiny dots, called pixels that are arranged in a rectangular form. The sharpness of the image depends upon the number of pixels.

There are two kinds of viewing screen used for monitors.

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

Cathode-Ray Tube (CRT) Monitor

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

A finite number of characters can be displayed on a screen at once. The screen can be divided into a series of character boxes - fixed location on the screen where a standard character can be placed. Most screens are capable of displaying 80 characters of data horizontally and 25 lines vertically.

There are some disadvantages of CRT

- Large in Size
- High power consumption

Flat-Panel Display Monitor

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

The flat-panel display is divided into two categories −

- **Emissive Displays** − Emissive displays are devices that convert electrical energy into light. For example, plasma panel and LED (Light-Emitting Diodes).
- **Non-Emissive Displays** − Non-emissive displays use optical effects to convert sunlight or light from some other source into graphics patterns. For example, LCD (Liquid-Crystal Device).

Printers

Printer is an output device, which is used to print information on paper.
There are two types of printers

- Impact Printers
- Non-Impact Printers

Impact Printers

Impact printers print the characters by striking them on the ribbon, which is then pressed on the paper.
Characteristics of Impact Printers are the following −

- Very low consumable costs
- Very noisy
- Useful for bulk printing due to low cost
- There is physical contact with the paper to produce an image

These printers are of two types

- Character printers
- Line printers

Character Printers
Character printers are the printers which print one character at a time.
These are further divided into two types:

- Dot Matrix Printer(DMP)
- Daisy Wheel

Dot Matrix Printer

In the market, one of the most popular printers is Dot Matrix Printer. These printers are popular because of their ease of printing and economical price. Each character printed is in the form of pattern of dots and head consists of a Matrix of Pins of size (5*7, 7*9, 9*7 or 9*9) which come out to form a character which is why it is called Dot Matrix Printer.

Advantages

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

Disadvantages

- Slow Speed
- Poor Quality

Daisy Wheel

Head is lying on a wheel and pins corresponding to characters are like petals of Daisy (flower) which is why it is called Daisy Wheel Printer. These printers are generally used for word-processing in offices that require a few letters to be sent here and there with very nice quality.

Advantages
- More reliable than DMP
- Better quality
- Fonts of character can be easily changed

Disadvantages
- Slower than DMP
- Noisy
- More expensive than DMP

Line Printers

Line printers are the printers which print one line at a time.

These are of two types

- Drum Printer
- Chain Printer

Drum Printer

This printer is like a drum in shape hence it is called drum printer. The surface of the drum is divided into a number of tracks. Total tracks are equal to the size of the paper, i.e. for a paper width of 132 characters, drum will have 132 tracks. A character set is embossed on the track. Different character sets available in the market are 48 character set, 64 and 96 characters set. One rotation of drum prints one line. Drum printers are fast in speed and can print 300 to 2000 lines per minute.

Advantages

- Very high speed

Disadvantages

- Very expensive
- Characters fonts cannot be changed

Chain Printer

In this printer, a chain of character sets is used, hence it is called Chain Printer. A standard character set may have 48, 64, or 96 characters.

Advantages

- Character fonts can easily be changed.
- Different languages can be used with the same printer.

Disadvantages

- Noisy

Non-impact Printers

Non-impact printers print the characters without using the ribbon. These printers print a complete page at a time, thus they are also called as Page Printers.

These printers are of two types

- Laser Printers
- Inkjet Printers

Characteristics of Non-impact Printers

- Faster than impact printers

- They are not noisy
- High quality
- Supports many fonts and different character size

Laser Printers

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

Advantages

- Very high speed
- Very high quality output
- Good graphics quality
- Supports many fonts and different character size

Disadvantages

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

Inkjet Printers

Inkjet printers are non-impact character printers based on a relatively new technology. They print characters by spraying small drops of ink onto paper. Inkjet printers produce high quality output with presentable features.

They make less noise because no hammering is done and these have many styles of printing modes available. Colour printing is also possible. Some models of Inkjet printers can produce multiple copies of printing also.

Advantages
- High quality printing
- More reliable

Disadvantages

- Expensive as the cost per page is high
- Slow as compared to laser printer

Memory Revisited

The memory stores data, instructions and results either temporarily or permanently. It plays vital role in the computer. So it is considered the "**brain**" of the computer. All memory devices can be divided into

1. **Primary storage devices :** Stores programs, data and results
   - (a) Main memory (RAM, ROM)
   - (b) Internal process memory (cache memory)
2. **Secondary storage devices :** Stores all Software, and other information
   - (a) Magnetic storage devices (Floppy disks)
   - (b) Optical storage devices (CDs, DVDs)
   - (c) Magneto-optical storage devices

(A). Main Memory: Main memory stores programs, data and results. There are two types of memory namely.

RAM (Random Access Memory) and **ROM** (Read Only Memory).

RAM: This memory is used for temporary storage of program, data and results when they are being executed by the computer. It is volatile in nature which means the program or data will be lost if power is switched off.

- Static RAM
- Dynamic RAM

ROM: This memory consists of predefined instructions defined by the manufacturer and are usually executed during system start up. It is non-volatile in nature which means instructions are not lost if power is switched off. It is read only memory.

- ROM
- PROM
- EPROM
- EEPROM
- FLASH MEMORY

b) **Cache Memory**

Cache memory is a small, fast and expensive memory that stores the copies of data that to be accessed frequently from the main memory. There are usually two types of cache memory found in the computer system

- Primary cache (L1 cache)
- Secondary cache (L2 cache)

2. Secondary Storage Devices

Computers need additional storage devices other than the main memory for two reasons. First, because the program, data or instructions are to be stored in a permanent area so that it can be retained when required and second, It can store more information than main memory.

Some of the frequently used secondary storage devices are hard disk, magnetic tape, floppy disk, zip disk, CD (Compact Disc) and DVD (Digital Video Disk). Each of these disks storage capacity varies from MB (Mega Bytes) to GB (Giga Bytes). Below is list showing storage capacities in terms of bytes and also showing the memory capacity of secondary storage devices.

- **Floppy Disks**
  - 3.5-inch disks store 1.44M of data
  - Must be formatted, organized in terms of Tracks and Sectors
- **Hard Disks**
  - **Sp**ins at 5,400 – 7,200 rpm (revolutions per minute)
  - Can store anywhere between 10G – 500GB+ of data
- **CD'sCompact Discs**
  - Available in a variety of formats—CD-ROM, CD-R, CD-RW
  - A typical CD holds about 650 ,720 MB of data
- **DVD's- Digital Video Disks**
  - Available as DVD-ROM, DVD-R, DVD-RW
  - Can hold 2 GB, 4.7 GB of data
- **Zip Drives --** high capacity floppy disk drive;
  - Zip disks can hold from 100 MB – 250 MB of data
- **USB Flash Drive**
  - Storage capacity between 32 MB – 4 GB

Data Representation Units:

• Bit	--	0 or 1 bit
• Byte	--	8-bits (1 word)
• Kilobyte	--	1 thousand bytes (1 KB = 1,000 bytes) - 2^{10} ~10^3 bytes
• Megabyte	--	1 million bytes (1MB =1,000,000 bytes)-2^{20} ~10^6 bytes
• Gigabyte	--	1 billion bytes (1GB=1,000,000,000bytes)-2^{30} ~10^9 bytes
• Terabyte	--	1 trillion bytes – 2^{40} ~ 10^{12} bytes
• Pet byte	--	1 quadrillion bytes – 2^{50} ~ 10^{15} bytes
• Exabyte	--	1 quintillion bytes
• Zeta byte	--	1 sextillion bytes
• Yota byte	--	1 septillion bytes

Organization of the Memory:

Memory of the computer is an ordered sequence of storage locations called **memory cells.** Each cell is identified by **unique address** which can be used to store or access the information. The **contents** of the memory cell can contain any type of data or instructions. The size of the memory cell usually varies from computer to computer. It is represented as **bytes**. Few computers use 1 byte for 1 memory cell, few use 2 bytes and some more use either 4 or 8 or 16 bytes or even more.

All information in the memory of the computer is usually stored as bits but for our convenience in the pictorial representation given below the contents of the memory cells are represented in a form which we can understand.

Memory cells

contents	-27	A	Add	0.02	...	...	..	..	266
Address	1001	1002	1003	1004					1020

Storage and Retrieval of information in Memory

To store a value the computer sets each bit of the memory cell to either 0 or 1. For Example to store A in a memory cell the bits are set as **01000001** if one byte of memory is allocated to memory cell. (i.e., for the character A the **ASCII** (American Standard Code for Information Interchange) code is 65 which is converted to bits as **01000001)**

To retrieve a value the computer copies the patterns of 0's and 1's stored in the memory cell to another storage area for processing.

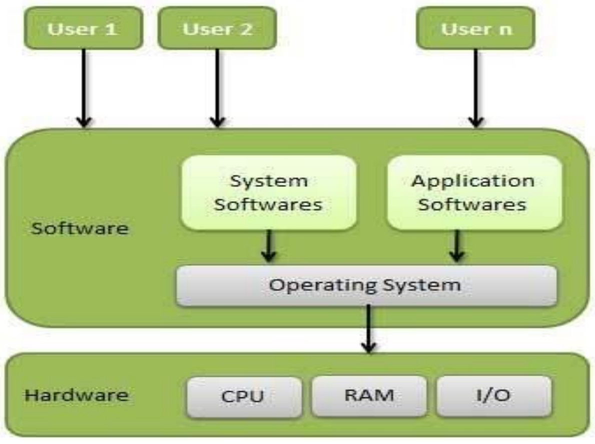
<u>Introduction to Operating systems</u>

An Operating System (OS) is an interface between a computer user and computer hardware. An operating system is software which performs all the basic tasks like file management, memory management, process management, handling input and output, and controlling peripheral devices such as disk drives and printers.

Some popular Operating Systems include Linux Operating System, Windows Operating System, VMS, OS/400, AIX, z/OS, etc.

Definition

An operating system is a program that acts as an interface between the user and the computer hardware and controls the execution of all kinds of programs.



Following are some of important functions of an operating System.

- Memory Management
- Processor Management
- Device Management
- File Management
- Security
- Control over system performance
- Job accounting
- Error detecting aids
- Coordination between other software and users

Memory Management

Memory management refers to management of Primary Memory or Main Memory. Main memory is a large array of words or bytes where each word or byte has its own address.

Main memory provides a fast storage that can be accessed directly by the CPU. For a program to be executed, it must in the main memory. An Operating System does the following activities for memory management.

- Keeps tracks of primary memory, i.e., what part of it are in use by whom, what part are not in use.
- In multiprogramming, the OS decides which process will get memory when and how much.
- Allocates the memory when a process requests it to do so.
- De-allocates the memory when a process no longer needs it or has been terminated.

Processor Management

In multiprogramming environment, the OS decides which process gets the processor when and for how much time. This function is called **process scheduling**. An Operating System does the following activities for processor management.

- Keeps tracks of processor and status of process. The program responsible for this task is known as **traffic controller**.
- Allocates the processor (CPU) to a process.
- De-allocates processor when a process is no longer required.

Device Management

An Operating System manages device communication via their respective drivers. It does the following activities for device management.

- Keeps tracks of all devices. Program responsible for this task is known as the **I/O controller**.
- Decides which process gets the device when and for how much time.
- Allocates the device in the efficient way.
- De-allocates devices.

File Management

A file system is normally organized into directories for easy navigation and usage. These directories may contain files and other directions.

An Operating System does the following activities for file management.

- Keeps track of information, location, uses, status etc. The collective facilities are often known as file system.
- Decides who gets the resources.
- Allocates the resources.
- De-allocates the resources.

Other Important Activities

Following are some of the important activities that an Operating System performs −

- **Security**: By means of password and similar other techniques, it prevents unauthorized access to programs and data.
- **Control over system performance**: Recording delays between request for a service and response from the system.
- **Job accounting**: Keeping track of time and resources used by various jobs and users.
- **Error detecting aids**: Production of dumps, traces, error messages, and other debugging and error detecting aids.
- **Coordination between other software's and users**: Coordination and assignment of compilers, interpreters, assemblers and other software to the various users of the computer systems.

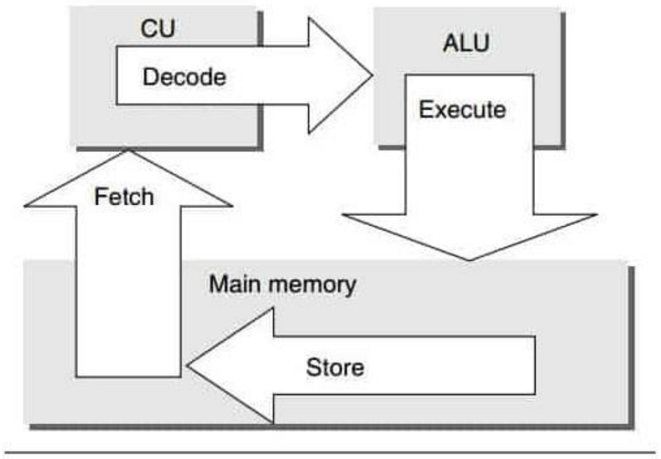
Operational Overview of a Cpu

Central Processing Unit (CPU) consists of the following features

- CPU is considered as the brain of the computer.
- CPU performs all types of data processing operations.
- It stores data, intermediate results, and instructions (program).
- It controls the operation of all parts of the computer.

CPU itself has following three components.

- Memory or Storage Unit
- Control Unit
- ALU(Arithmetic Logic Unit)



A simplified view of an instruction cycle

Memory or Storage Unit

This unit can store instructions, data, and intermediate results. This unit supplies information to other units of the computer when needed. It is also known as internal storage unit or the main memory or the primary storage or Random Access Memory (RAM).

Its size affects speed, power, and capability. Primary memory and secondary memory are two types of memories in the computer. Functions of the memory unit are −

- It stores all the data and the instructions required for processing.
- It stores intermediate results of processing.
- It stores the final results of processing before these results are released to an output device.
- All inputs and outputs are transmitted through the main memory.

Control Unit

This unit controls the operations of all parts of the computer but does not carry out any actual data processing operations.

Functions of this unit are

- It is responsible for controlling the transfer of data and instructions among other units of a computer.
- It manages and coordinates all the units of the computer.
- It obtains the instructions from the memory, interprets them, and directs the operation of the computer.
- It communicates with Input/output devices for transfer of data or results from storage.
- It does not process or store data.

ALU (Arithmetic Logic Unit)

This unit consists of two subsections namely,

- Arithmetic Section
- Logic Section

Arithmetic Section

Function of arithmetic section is to perform arithmetic operations like addition, subtraction, multiplication, and division. All complex operations are done by making repetitive use of the above operations.

Logic Section

Function of logic section is to perform logic operations such as comparing, selecting, matching, and merging of data.

Introduction to Programming

C is a high-level structured oriented programming language, used in general purpose programming, developed by Dennis Ritchie at AT&T Bell Labs, in USA between 1969 and 1973.

Some Facts about C Programming Language

- In 1988, the American National Standards Institute (ANSI) had formalized the C language.
- C was invented to write UNIX operating system.
- C is a successor of 'Basic Combined Programming Language' (BCPL) called B language.
- Linux OS, PHP, and MySQL are written in C.
- C has been written in assembly language.

Uses of C Programming Language

In the beginning, C was used for developing system applications, e.g. :

- Database Systems
- Language Interpreters
- Compilers and Assemblers
- Operating Systems
- Network Drivers
- Word Processors

C has Become Very Popular for Various Reasons

- One of the early programming languages.
- Still, the best programming language to learn quickly.
- C language is reliable, simple and easy to use.
- C language is a structured language.
- Modern programming concepts are based on C.
- It can be compiled on a variety of computer platforms.
- Universities preferred to add C programming in their courseware.

Features of C Programming Language

- C is a robust language with a rich set of built-in functions and operators.
- Programs written in C are efficient and fast.
- C is highly portable, programs once written in C can be run on other machines with minor or no modification.
- C is a collection of C library functions; we can also create our function and add it to the C library.
- C is easily extensible.

Advantages of C

- C is the building block for many other programming languages.
- Programs written in C are highly portable.
- Several standard functions are there (like in-built) that can be used to develop programs.
- C programs are collections of C library functions, and it's also easy to add own functions to the C library.
- The modular structure makes code debugging, maintenance and testing easier.

Disadvantages of C

- C does not provide Object Oriented Programming (OOP) concepts.
- There are no concepts of Namespace in C.
- C does not provide binding or wrapping up of data in a single unit.
- C does not provide Constructor and Destructor.

Programs and Programming

A computer can neither think nor make a decision on its own. In fact, it is not possible for any computer to independently analyze a given data and find a solution on its own. It needs a program which will convey what is to be done.

A **program** is a set of logically related instructions that is arranged in a sequence that directs the computer in solving a problem.

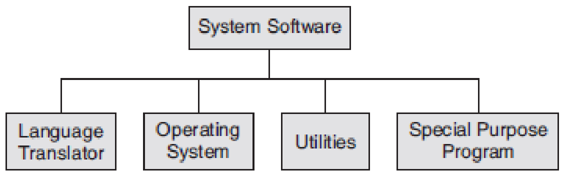
Computer Software refers to a set of computer programs that are required to enable the hardware components work together to solve a problem. Computer Program is set of instructions written in a programming language that tells the computer how to accomplish a task. Computer Instruction is a simple command given to solve a problem.

Types of Software's:

- System Software
- Application Software
- Programming Languages

System Software:

It is used to control and co-ordinate hardware components and manage their tasks. And the available system software is.



(a) System Management /Support software: Used to manage hardware and software

Example: OS, Device Drivers, Utility programs (Anti-Virus)

Operating System (OS):

It tells the computer how to perform the functions of loading, storing and executing an application and how to transfer data. It is a collection of programs that act as an interface between the user of the computer and computer hardware

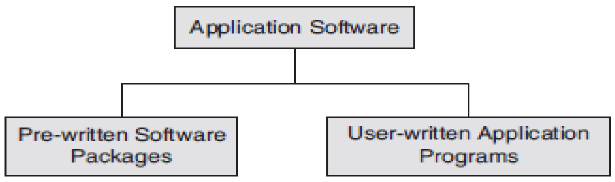
Examples: DOS, UNIX. (Command line interface), Windows (graphical user interface).

(b) System Development/Programming Software: Used for developing and executing application
Software

Example: Language Translator, Linker, Compiler, Interpreter, Debugger

Application Software:

It is used to develop the user required programs that tell a computer how to produce information. Commonly used applications are Word processing, Spreadsheet, Database, Presentation, Financial, Email, and Taxing.


Application Software
Pre-written Software Packages
User-written Application Programs

(A). Standard Application Software:

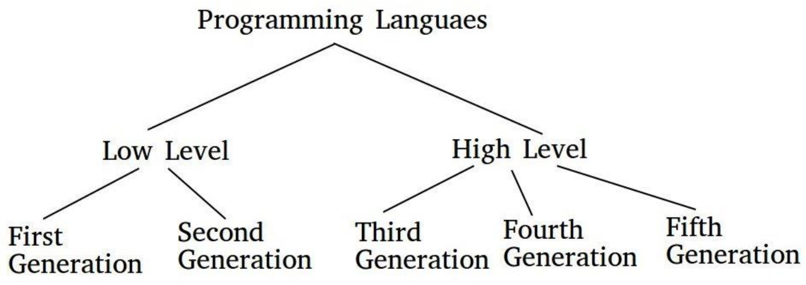
It is used to develop applications like word, power point, Database manager, Web browser etc.., which are used for general processing tasks.

(B). Unique Application Software:

It is used to develop applications of specific needs like Tally for Accounts purpose.

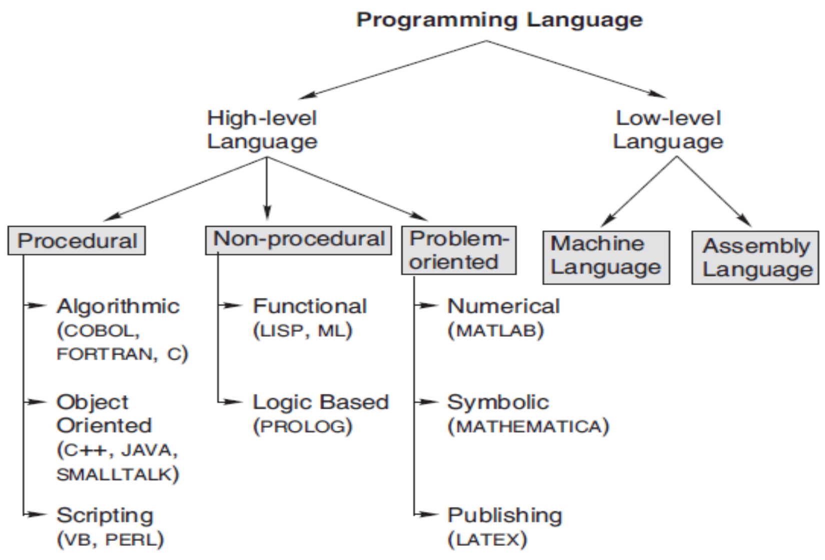
Computer Programming Languages/Classification of Programming languages:

To write a program for a computer, we must use a computer language. A vocabulary and set of grammatical rules for instructing a computer to perform specific tasks. Over the years computer languages have evolved from machine language to natural languages.



Types of Programming Languages:

- **Low Level Languages**
  - Machine Level Language
  - Assembly Level Language
- **High Level Language**



1) Low Level Languages

(i) **Machine Level Language:(1940s) --First Generation** :

This is the language which the machine can understand. Statements will be written as combination of 0's and 1's. This language is not standardized and has the difficulty of writing separately for each machine. **Example:** 100101111001 010101000111

(ii) Assembly Level Language: (1950s)--Second Generation :

This is the language in which the programming statements are written using mnemonic codes for performing operations. For example MUL A, where MUL is the mnemonic code signifying the multiplication operation.

```
Example:LOAD   BASEPAY
         ADD    OVERPAY
      STORE GROSSPAY
```

Disadvantages of assembly language

- ✓ Assembly language is specific to particular machine architecture, i.e., machine dependent. Assembly languages are designed for a specific make and model of a microprocessor. This means that assembly language programs written for one processor will not work on a different processor if it is architecturally different. That is why an assembly language program is not portable.
- ✓ Programming is difficult and time consuming.
- ✓ The programmer should know all about the logical structure of the computer.

High Level Language: (1960s) -- ThirdGeneration:

This is the language in which the programming statements look very much similar to that of English. It takes the advantage of running programs on any machine. Examples of such languages are C, C++, Java, COBOL, FORTRAN, Small talk, LISP, PROLOG etc.

Example: gross Pay = base Pay + overtime Pay

Advantages of high-level programming languages

- ✓ **Readability:** Programs written in these languages are more readable than those written in assembly and machine languages run on different machines with little or no change. It is, therefore, possible to exchange software, leading to creation of program libraries.

- ✓ **Easy debugging:** Errors can be easily detected and removed.

- ✓ **Ease in the development of software:** since the commands of these programming languages are closer to the English language, software can be developed with ease. High-level languages are also called third generation languages (3GLs).

Algorithmic languages: These are high-level languages designed for forming convenient expression of procedures, used in the solution of a wide class of problems. In this language, the programmer must specify the steps the computer has to follow while executing a program. Some of languages that fall in the category are C, COBOL, and FORTRAN.

Object-oriented language: The basic philosophy of object oriented programming is to deal with objects rather than functions or subroutines as in strictly algorithmic languages.

Objects are self-contained modules that contain data as well as the functions needed to manipulate the data within the same module. In a conventional programming language, data and subroutines or functions are separate. In object oriented programming, subroutines as well as data are locally defined in objects. The difference affects the way a programmer goes about writing a program as well as how information is represented and activated in the computer.

- Abstraction
- Encapsulation And Data Hiding
- Polymorphism
- Inheritance
- Reusable Code
- C++, Java, Smalltalk, Etc. Are Examples Of Object Oriented Languages.

Scripting languages: These languages assume that a collection of useful programs, each performing a task, already exists. It has facilities to combine these components to perform a complex task. A scripting language may thus be thought of as a glue language, which sticks a variety of components together. One of the earliest scripting languages is the UNIX shell. Now there are several scripting languages such as VB script and Perl.

Problem-oriented Languages

These are high-level languages designed for developing a convenient expression of a given class of problems.

Non-procedural Languages

Functional (applicative) languages: These function al languages solve a problem by applying a set of functions to the initial variables in specific ways to get the answer. The functional programming style relies on the idea of function application rather than on the notion of variables and assignments. A program written in a function al language consists of function calls together with arguments to functions. LISP, ML, etc. are examples of function al languages.

Logic-based programming language A logic program is expressed as a set of atomic sentences, known as fact, and horn clauses, such as if-then rules. A query is then posed. The execution of the program now begins and the system tries to find out if the answer to the query is true or false for the given facts and rules. Such languages include PROLOG.

More about System Development/Programming Software:

It provides tools to assist the programmer in writing computer programs. It includes Compiler, Interpreter, Assembler, Linker, Text editor etc.

Compiler: It is a type of translator/program that translates high level language program to machine level language program at a time. It works quickly compared to interpreter

Example: C, C++, FORTRAN, COBOL, Pascal

Interpreter: It is a type of translator/program that translates high level language statement to machine level language statement one by one. It works slowly compared to compiler

Example: basic, visual basic.

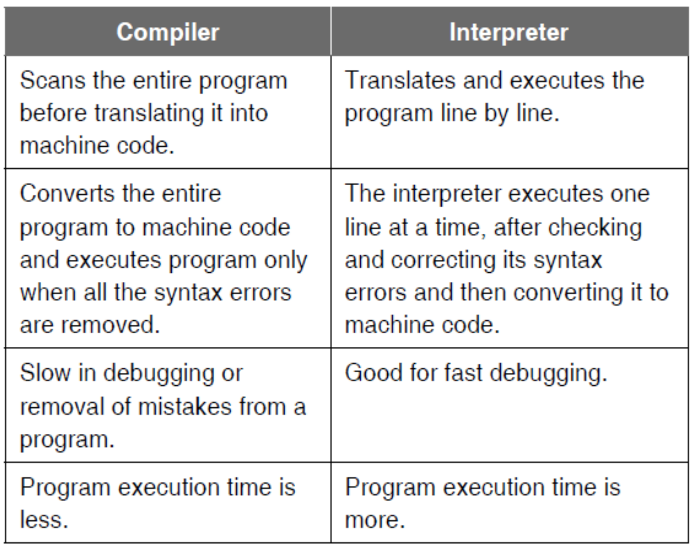
Assembler: It is a type of translator/program that translates assembly level language program to machine level language program.**Example:** masm, tasm.

Loader: This program copies the executable statements to memory and initiates the execution of Instructions.

Linker: This program combines the pre-fabricated functions to the object file to create an executable Machine file.

IDE (Integrated Development Environment): It refers to the software that provides support for editing the program, compiling, linking and loading.

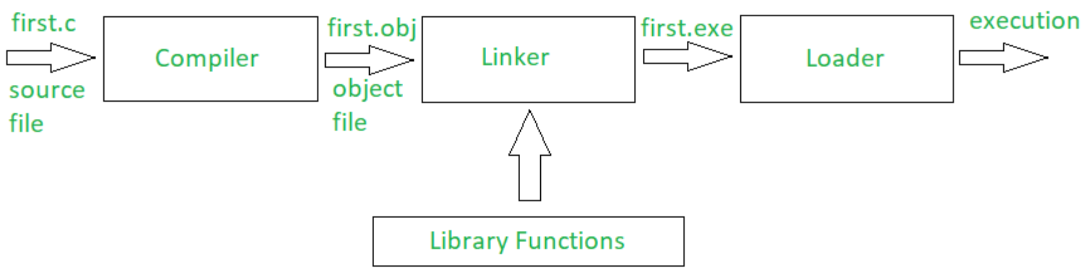
Differences between a compiler and an Interpreter



Compiler	Interpreter
Scans the entire program before translating it into machine code.	Translates and executes the program line by line.
Converts the entire program to machine code and executes program only when all the syntax errors are removed.	The interpreter executes one line at a time, after checking and correcting its syntax errors and then converting it to machine code.
Slow in debugging or removal of mistakes from a program.	Good for fast debugging.
Program execution time is less.	Program execution time is more.

Program execution

Whenever a C program file is compiled and executed, the compiler generates some files with the same name as that of the C program file but with different extensions. So, what are these files and how are they created?



Every file that contains a C program must be saved with '.c' extension. This is necessary for the compiler to understand that this is a C program file. Suppose a program file is named, first.c. The file first.c is called the source file which keeps the code of the program. Now, when we compile the file, the C compiler looks for errors. If the C compiler reports no error, then it stores the file as an .obj file of the same name, called the object file. So, here it will create the first.obj. This .obj file is not executable. The process is continued by the Linker which finally gives an .exe file which is executable.

Generation of Languages:

There are five generation of Programming languages. They are:

First Generation Languages :

These are low-level languages like machine language.

Second Generation Languages:

These are low-level assembly languages used in kernels and hardware drives.

Third Generation Languages:

These are high-level languages like C, C++, Java, Visual Basic and JavaScript.

Fourth Generation Languages:

- ✓ These are languages that consist of statements that are similar to statements in the human language.

- ✓ These are used mainly in database programming and scripting. Example of these languages includes Perl, Python, Ruby, SQL, Mat Lab (Matrix Laboratory).

Fifth Generation Languages:

- ✓ These are the programming languages that have visual tools to develop a program. Examples of fifth generation language include Mercury, OPS5, and Prolog.

- ✓ The first two generations are called low level languages. The next three generations are called high level languages.

Fourth Generation (Programming) Language (4GL)

4GLs are more programmer-friendly and enhance programming efficiency with usage of English-like words and phrases, and when appropriate, the use of icons, graphical interfaces and symbolical representations. The key to the realization of efficiency with 4GLs lies in an appropriate match between the tool and the application domain. Additionally, 4GLs have widened the population of professionals able to engage in software development.

Many 4GLs are associated with databases and data processing, allowing the efficient development of business-oriented systems with languages that closely match the way domain experts formulate business rules and processing sequences. Many of such data-oriented 4GLs are based on the Structured Query Language (SQL), invented by IBM and subsequently adopted by ANSI and ISO as the standard language for managing structured data.

Most 4GLs contain the ability to add 3GL-level code to introduce specific system logic into the 4GL program.

The most ambitious 4GLs, also denoted as Fourth Generation Environments, attempt to produce entire systems from a design made in CASE tools and the additional specification of data structures, screens, reports and some specific logic

Advantages of 4GLs

- Programming productivity is increased. One line ofa 4GL code is equivalent to several lines of a 3GLcode.
- System development is faster.
- Program maintenance is easier.
- End users can often develop their own applications.
- Programs developed in 4GLs are more portable than those developed in other generation languages.
- Documentation is of improved order because most4GLs are self-documenting.

3GL vs 4GL

3GL	4GL
Meant for use by professional programmers	May be used by non-professional programmers as well as by professional programmers.
Requires specifications of how to perform a task	Requires specifications of what task to perform.
All alternatives must be specified	System determines how to perform the task.
Execution time is less	Default alternatives are built-in. User need not specify these alternatives.
Requires large number of procedural instructions Code may be difficult to read, understand, and maintain by the user	Requires fewer instructions.
Typically, file oriented	Difficult to debug

Fifth Generation (Programming) Language (5GL)

Natural languages represent the next step in the development of programming languages belonging to fifth generation languages. Natural language is similar to query language, with one difference: it eliminates the need for the user or programmer to learn a specific c vocabulary, grammar, or syntax. The text of a natural language statement resembles human speech closely. Infact, one could word a statement in several ways, perhaps even misspelling some words or changing the order of the words, and get the same result. Natural language takes the user one step further away from having to deal directly and in detail with computer hardware and software. These languages are also designed to make the computer
Smarter—that is, to simulate the human learning process. Natural languages already available for micro computers include CLOUT, Q & A, and SAVY RETRIEVER (for use with databases) and HAL (Human Access Language) for use with LOTUS.

PROLOG (acronym for Programming Logic) is an example of a Logical Programming Language. It uses a form of mathematical logic (predicate calculus) to solve queries on a programmer-given database of facts and rules.

Structured Programming Concept

Structured programming is a subset of procedural programming. It is also known as modular programming. Corrado Bohm and GuiseppeJacopini first suggested **structured programming language**. It is a logical programming method that is considered a precursor to object-oriented programming (OOP). Its main purpose to enforce a logical structure on the program being written to make it more efficient and easier to understand and modify.

The **structured programming language** allows a programmer to code a program by dividing the whole program into smaller units or modules. Structured programming is not suitable for the development of large programs and does not allow re usability of any set of codes. It is a programming paradigm aimed at improving the quality, clarity, and access time of a computer program by the use of subroutines, block structures, for and while loops.

Advantages of Structured Programming Language

- Structured programming is user-friendly and easy to understand.
- In this programming, programs are easier to read and learn.
- It avoids the increased possibility of data corruption.
- The main advantage of structured programming is reduced complexity.
- Increase the productivity of application program development.
- Application programs are less likely to contain logic errors.
- Errors are more easily found.
- It is easier to maintain.
- It is independent of the machine on which it is used, i.e. programs developed in high-level languages can be run on any computer.

Structure of C programming

- Documentations (Documentation Section)
- Pre-processor Statements (Link Section)
- Global Declarations (Definition Section)
- The main() function
  - Local Declarations
  - Program Statements & Expressions
- User Defined Functions

Algorithms

An algorithm is a sequence of steps to solve a particular problem. Algorithms are universal. The algorithm you use in C programming language is also the same algorithm you use in every other language.

Characteristics of Algorithm:

1. **Input**: The inputs used in an algorithm must come from a specified set of elements, where the amount and type of inputs are specified.
2. **Output**: The algorithm must specify the output and how it is related to the input.
3. **Definiteness**: The steps in the algorithm must be clearly defined and detailed.
4. **Effectiveness**: The steps in the algorithm must be doable and effective.
5. **Finiteness**: The algorithm must come to an end after a specific number of steps.

Types of algorithm

Well there are many types of algorithm but the most fundamental types of algorithm are:

1. Recursive algorithms
2. Dynamic programming algorithm
3. Backtracking algorithm
4. Divide and conquer algorithm
5. Greedy algorithm
6. Brute Force algorithm
7. Randomized algorithm

Different Ways of Stating Algorithms

Algorithms may be represented in various ways. There are four ways of stating algorithms.

- Step-form
- Pseudo-code
- Flowchart

Step-form:

In the step form representation, the procedure of solving a problem is stated with written statements. Each statement solves a part of the problem and these together complete the solution.

The step-form uses just normal language to define each procedure. Every statement, that defines an action, is logically related to the preceding statement.

Pseudo code:

Pseudo code is not an actual programming language. It's simply an implementation of an algorithm in the form of text written in plain English. It has no syntax like any of the programming language and thus can't be compiled or interpreted by the computer.

Flowchart:

Flowchart is a graphically oriented representation forms. They use symbols and language to represent sequence, decision, and repetition actions. Only the flowchart method of representing the problem solution has been explained with several examples.

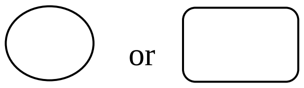
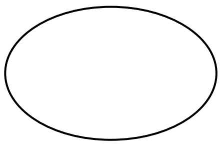
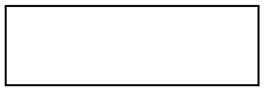
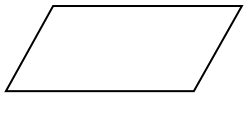
Example:

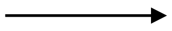
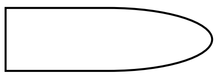
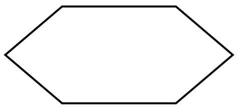
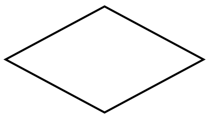
1. Start
2. Declare variables num1,num2 and sum.
3. Read values num1 and num2.
4. Add num1 and num2 and assign the result to sum.
   Sum=num1+num2
5. Display Sum
6. Stop.

Flow Chart:

A Flowchart can be defined as the pictorial/diagrammatic/visual representation of a process, which describes the sequence and flow of execution control of steps described in the algorithm/process. It increases the understand ability of the process

There are different symbols used to indicate the operational steps of an algorithm.

SYMBOL NAME	SYMBOL	SYMBOL PURPOSE
Rounded Rectangle or Flat oval or Ellipse		Indicates the beginning (start) and ending (stop) of the flowchart
Small Circle		Indicates connector. It is used in situations whenever it is needed to join two parts of the flowchart.
Rectangle		Indicates the process. It is used to represent computation or logic or assignment statements of an algorithm
Parallelogram		Indicates Input/output symbol. It is used to represent the step of algorithm whenever data is to be read or displayed

Lines with arrow mark		Indicates the flow lines. It indicates the direction of flow (Down, Up, Left, Right) .
Symbol of "AND" gate		Indicates delay symbol. It is used for adding delay to the process step.
Hexagon		Indicates Loop. It is used when iteration steps of algorithm is to be represented
Diamond symbol or Trapezium		Indicates decision symbol. It is used to represent selection steps of an algorithm.

Flowchart Design Rules:

1. It must begin with "Start" and end with "Stop" symbol.
2. The process flow should be either form top-to-bottom or bottom-to-top.
3. The instructions specified in flow chart must be crisp and concise.
4. Two arrows must never intersect or cross each other.
5. A process symbol must have only one input and one output arrow.

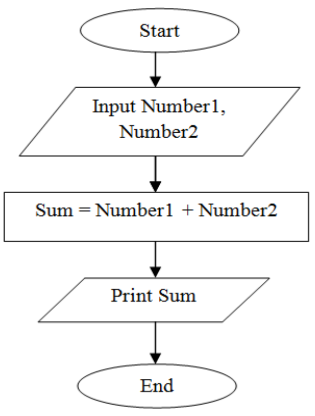
Advantages of Flowcharts:

1. It helps in understanding the flow of program execution control in a easy way.
2. Developing the program code by referring its flowchart is easier one.
3. It helps in avoiding semantic errors.
4. It is easier to understand the pictorial representation than sequence of statements.

Disadvantages of Flowcharts:

1. Flowchart works well for small program design.
2. For large programs, the flowchart become very complex and confusing
3. Modification of a flowchart is difficult and requires almost entire work.
4. Excessive use of connectors in the flowchart creates confusion.
5. As flow chart consists of more symbols, it becomes little tedious to design flowchart.

Flow Chart for Sum of two numbers:



Strategy for Designing Algorithms:

Now that the meaning of algorithm and data has been understood, strategies can be devised for designing algorithms. The following is a useful strategy.

1. Identify the outputs needed.

This includes the form in which the outputs have to be presented. At the same time, it has to be determined at what intervals and with what precision the output data needs to be given to the user.

2. Identify the input variables available

This activity considers the specifi c inputs available for this program, the form in which the input variables would be available, the availability of inputs at different intervals, the ways in which the input would be fed to the transforming process.

3. Identify the major decisions and conditions

This activity looks into the conditions imposed by the need identified and the limitations of the environment in which the algorithm has to be implemented.

4. **Identify the processes required to transform inputs into required outputs.**

This activity identifies the various types of procedures needed to manipulate the inputs, within the bounding conditions and the limitations mentioned in step 3, to produce the needed outputs.

5**. Identify the environment available.**

This activity determines the kind of users and the type of computing machines and software available for implementing the solution through the processes considered in steps.

Tracing an Algorithm to Depict logic:

An algorithm is a collection of some procedural steps that have some precedence relation between them. Certain procedures may have to be performed before some others are performed. Decision procedures may also be involved to choose whether some procedures arranged one after other are to be executed in the given order or skipped or implemented repetitively on fulfillment of conditions arising out of some preceding manipulations. Hence, an algorithm is a collection of procedures that results in providing a solution to a problem. Tracing an algorithm primarily involves tracking the outcome of every procedure in the order they are placed. Tracking in turn means verifying every procedure one by one to determine and confirm the corresponding result that is to be obtained. This in turn can be traced to offer an overall output from the implementation of the algorithm.

The above method shows how the logic of an algorithm, planned and represented by a tool like the flowchart, can be verified for its correctness. This technique, also referred to as deskcheck or dry run, can also be used for algorithms represented by tools other than the flowchart.

Specification for Converting Algorithms into Programs:

By now, the method of formulating an algorithm has been understood. Once the algorithm, for solution of a problem, is formed and represented using any of the tools like Step-form or flowchart or pseudo code, etc., it has to be transformed into some programming language code. This means that a program, formed by a sequence of program instructions belonging to a programming language, has to be written to represent the algorithm that provides a solution to a problem.

Code the algorithm into a program : Understand the syntax and control structures used in the language that has been selected and write the equivalent program instructions based upon the algorithm that was created.

Each statement in an algorithm may require one or more lines of programming code.

Desk-check the program: Check the program code by employing the desk-check method and make sure that the sample data selected produces the expected output.

Evaluate and modify, if necessary, the program: Based on the outcome of desk-checking the program, make program code changes, if necessary, or make changes to the original algorithm, if need be.

Do not reinvent the wheel: If the design code already exists, modify it, do not remake it.

****

UNIT-II

Unit 2

Introduction to computer problem solving: Introduction, the problem-solving aspect, top down design, implementation of algorithms, the efficiency of algorithms, the analysis of algorithms.

Fundamental algorithms: Exchanging the values of two variables, counting, summation of a set of numbers, factorial computation, sine function computation, generation of the Fibonacci sequence, reversing the digits of an integer.

Introduction:

Computer problem solving can be summed up in one word it is **demanding**, in this process required much careful planning, logical precision, persistence, and attention to detail. At the same time it can be a challenging, exciting, and satisfying experience with considerable room for personal creativity and expression.

This set of instructions is called program, a program may also be thought of as an algorithm expressed in a programming language. An algorithm is independent of any programming language. To obtain the computer solution to a problem, have to supply the input or data. The program takes the input and manipulates it according to its instructions and eventually produces an output which represents the computer solution to the problem.

An algorithm consists of a set of explicit and unambiguous finite steps which, when carried out for a given set of initial conditions, produce the corresponding output and terminate in a finite time. After studying even a small sample of computer problems it soon becomes obvious that the conscious deapth of understanding.

The Problem-Solving Aspect:

- ✓ Specify the problem
- ✓ Analysis Phase
- ✓ Designing Phase
- ✓ Coding Phase
- ✓ Testing Phase
- ✓ Maintenance Phase

Specify the problem:

This step involves trying to understand the statement of the given problem.

Analysis Phase:

The analysis phase involves identifying the various inputs that are required and the output that is expected by the problem.

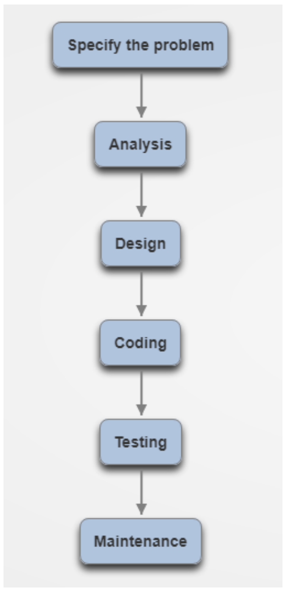
Designing Phase:

The design phase involves a sequence of steps that will allow us to take the given input to get the required output. A flow chart or some other tool can be used to pictorially represent the design step by step.

Coding Phase: The coding phase involves translating the steps identified in the design phase into a machine readable code (program) which is executed to produce the required output. A program is written in a programming language (machine readable code).

Testing Phase:
The testing phase involves verifying if the output received in the coding phase is the required output using the given input.

Maintenance Phase:
A software program that is developed and tested made available to customers. Any change required to this program involves adding small updates. This is the maintenance phase.



Top down Design:

The primary goal in computer problem solving is an algorithm which is capable of being implemented as a correct and efficient computer program. Once we have defined the problem to be solved and we have at least an idea on how to solve it, we can design algorithms using different techniques.

Top-down design or **stepwise refinement** is one such technique for designing algorithms. Top-down design is a strategy that we can apply to take the solution of a computer problem from a vague outline to a precisely defined algorithm and program implementation. It allows us to build our solutions to a problem in a stepwise fashion.

1) Breaking a problem into sub problems

the top-down design suggests that we take the general statements that we have about the solution, and break them down into a set of more precisely defined subtasks. The process of repeatedly breaking a task down into subtasks should continue until we eventually end up with subtasks that can be implemented as program statements.

2) Choice of a suitable data structure

All programs operate on data and consequently, the way the data is organized can have a profound effect on every aspect of the final solution. The things we must be aware of in setting up data structures are: can it be easily searched, updated, can we recover to an earlier state of the computation, etc.,

Implementation of Algorithms

For a properly designed algorithm, the path of execution should flow in a straight line i.e from top to bottom. Programs implemented in this way are easy to understand, debug and modify.

Use of Procedures to emphasize modularity: It is usually helpful to modularize the program along the lines of Top-down design. This practice allows us to implement a set of independent procedures to perform specific and well-defined tasks.

Choice of Variable names: To make the program meaningful and easier to understand, the names of the variables and constants should be appropriate. For example, if we have to make the changes on the day of the week it is better to use a variable name as **day** instead of using single letter **a** or **b**. A clear definition of all variables and constants at the start of each procedure can also be helpful.

Documentation of Programs: We need to associate brief and accurate comments for each program. The program must specify exactly what responses it requires from the user during execution. A good programming practice is to always write programs so that they can be executed and used by other people.

Program Testing: While testing the program, all the boundary conditions should be verified. It is often not possible to write programs that handle all input conditions that may be given to a particular problem. It is always a good practice to write an algorithm that should satisfy a wide range of possible inputs.

Analysis of Algorithms: Good algorithms generally possess the following qualities and capabilities:

- They are simple and easily understandable by others
- They can be modified easily if necessary
- They have correct and clear defined solutions
- They are economical in the use of computer time, storage and peripherals
- They should run independent of computer
- The solution should be pleasing and satisfied by the designer.

Fundamental algorithms

Write a c program to exchange of two numbers without using third variable

Algorithm:

```
Step 1: Start
Step 2: Read the value of Number1, Number2
Step 3: Number1= Number1 + Number2
Step 4: Number2= Number1 - Number2
Step 5: Number1= Number1 - Number2
Step 6: PRINT the value of Number1 and Number2
Step 7: Stop
```

Source Code:-

```
#include<stdio.h>
#include<conio.h>
void main()
{
	int a,b;
	clrscr();
	printf("\nEnter two numbers :");
	scanf("%d%d",&a,&b);
	printf("\n Before swapping, values are a=%3d and b=%3d",a,b);
	a=a+b;
	b=a-b;
	a=a-b;
	printf("\n After swapping, values are a=%3d and b=%3d",a,b);
	getch();
}
```

Input:-

```
Enter two numbers: 7 9
```

Output:-

```
 Before swapping, values are a=7 and b=9
 After swapping, values are a=9 and b=7
```

Write a c program to exchange of two numbers using third variable

Algorithm: for using a third variable

```
    Step 1: Start
    Start 2: READ num1, num2
    Start 3: temp = num1
    Start 4: num1 = num2
    Start 5: num2 = temp
    Start 6: PRINT num1, num2
    Start 7: Stop
```

Source Code:-

```
#include<stdio.h>
#include<conio.h>
void main()
{
	int a,b,temp;
	clrscr();
	printf("\nEnter two numbers :");
	scanf("%d%d",&a,&b);
	printf("\n Before swapping, values are a=%3d and b=%3d",a,b);
	temp=a;
	a=b;
	b=temp;
	printf("\n After swapping, values are a=%3d and b=%3d",a,b);
	getch();
}
```

Input:-

```
Enter two numbers: 7 9
```

Output:-

```
 Before swapping, values are a=7 and b=9
 After swapping, values are a=9 and b=7
```

Write a c program to count the digits in a given number.

```
Step 1: Start
Start 2: READ number
Start 3: Initilize the count ←0
Start 4: repert step 5 untill number become 0 other wise goto step 7
Start 5: devide number with 10
Start 6: count←count+1
Start 7: write count
Start 8: Stop
```

Source Code:

```
#include<stdio.h>
#include<conio.h>
void main()
{
Long long n;
Int count=0;
Printf(enter the integer);
Scanf("%d",&n);
While(n>0)
{
n=n/10;
count++;
}
Printf("number of digitsis:%d",count);
}
```

Input:-

```
Enter an integer: 755
```

Output:-

```
 Number of digits is:3
```

Write a program to find the sum of the digits of a number

Algorithm:

```
Step 1: Get number by user
Step 2: Get the modulus/remainder of the number
Step 3: sum the remainder of the number
Step 4: Divide the number by 10
Step 5: Repeat the step 2 while number is greater than 0.
```

Source Code:-

```
#include<stdio.h>
#include<conio.h>
void main()
{
        int n ,r ,sum=0;
        clrscr();
        printf("\n Enter the number : ");
        scanf("%d",&n);
        while(n!=0)
         {
                r=n%10;
                sum=sum+r;
                n=n/10;
        }
        printf("\n Sum of the digits  is = %4d",sum);
        getch();
 }
```

Input:

```
  Enter the number: 256
```

Output:

```
Sum of the digits is = 13
```

Write a program to compute the factorial of a given number

Algorithm:

```
Step 1: Start
 Start 2: Read n
 Start 3: Initialize counter variable i and fact to 1
 Start 4: if condition is true go to step 5 otherwise go to step 7
 Start 5: calculate fact = fact * i
 Start 6: increment counter variable i and go to step 4
 Start 7: Write fact.
 Start 8: Stop
```

Source Code:-

```
#include<stdio.h>
#include<conio.h>
void main()
{
                int n,i;
                unsigned long long fact=1;
                clrscr();
                printf("\n Enter a number:");
                scanf("%d",&n);
                if(n<0)
                {
                Printf("factorial for a negative number doesn't exists");
                }
                else
                {
                for(i=1; i<=n; i++)
                {
                  fact*=i;// fact=fact*i;
                   i++;
                }
                printf("Factorial of %d is = %llu",n,fact);
                getch();
}
```

Input

```
 Enter a number:6
```

Output

```
Factorial of  6 is =  720
```

Write a program to compute the sine function computation

Algorithm:

```
    Step1: Start
    Step 2: Initialize int i,n
    Step 3: Initialize double sum,x,t
    Step 4: Read x and n values
    Step 5: find the x value using formula.
            X=x*π/180 // 3.14159/180
    Step 6: Check the condition using for loop
    Step 7: Compute the sin function t-> (t*(-1)*x*x)/(2*i*(2*i+1))
    Step 8: Calculate the sin function using sum=sum + t
    Step 9: Display sum
    Step 10: Stop
```

Source Code:

```
#include<stdio.h>
#include<conio.h>
void main()
{
  int i,n;
float x,sum,t;
clrscr();
printf("enter the value of x:");
scanf("%f",&x);
printf("enter the value of n:");
scanf("%d",&n);
x=x*3.14159/180;
t=x;
sum=x;
for(i=1;i<=n;i++)
{
t=(t*(-1)*x*x)/(2*i*(2*i+1));
sum=sum+t;
}
printf("the value of sin(%f)=%0.4f",x,sum);
}
```

Input

```
 Enter the value of x:45
Enter the value of n:4
```

Output

```
The value of sin (0.785398) =0.7071
```

Write a program to generate Fibonacci numbers in the given range.

Algorithm:

```
Step 1: Start
Step 2: Declare and initialize the necessary variables
Step 3: Check the condition, if the condition is true enter into loop
Step 4:Display F value till the condition get false
Step 5: Stop
```

Source Code:

```
#include<stdio.h>
#include<conio.h>
void main()
{
        int n,i,f1=0,f2=1,f;
        clrscr();
        printf(" \n Enter how many fibonacci numbers you want to print : ");
        scanf("%d",&n);
        printf("\n Fibonacci series is :\n");
        printf("%d\t%d",f1,f2);
        for(i=3;i<=n;i++)
        {
                f=f1+f2;
                printf("\t%d",f);
                f1=f2;
                f2=f;
        }
        getch();
}
```

Input:

```
 Enter how many Fibonacci numbers you want to print: 8
```

Output:-

```
 Fibonacci series is:
 0 1 1   2   3 5 8 13
```

Write a program to reverse the digits of a number.

Algorithm:

```
Step 1: Start
Step 2: Declare and initialize the necessary variables
Step 3: Enter the number to reverse
Step 4: Check the condition if the condition is true enter into loop
Step 5: Compute the following logic
        reverse=reverse*10;
        reverse=reverse+n%10;
                n=n/10;
Step 6: Display reverse value
Stop 7: Stop
```

Source Code:

```
#include<stdio.h>
#include<conio.h>
void main()
{
        int n,reverse=0;
        clrscr();
        printf("\n Enter a number :");
        scanf("%d",&n);
        While (n! =0)
        {
        reverse=reverse*10;
        reverse=reverse+n%10;
                n=n/10;
        }
        printf("\n The reverse of given number is:%d", reverse);
}
```

Input:

```
Enter a number: 325
```

Output:

```
 The reverse of given number is: 523
```

****

UNIT-III

Unit 3

History of C language is interesting to know. Here we are going to discuss a brief history of the c language.

C programming language was developed in 1972 by Dennis Ritchie at bell laboratories of AT&T (American Telephone & Telegraph), located in the U.S.A.

- **Dennis Ritchie** is known as the **founder of the c language**.
- It was developed to overcome the problems of previous languages such as B, BCPL, etc.
- Initially, C language was developed to be used in **UNIX operating system**. It inherits many features of previous languages such as B and BCPL.
- Let's see the programming languages that were developed before C language.

Language	**Year**	**Developed By**
Algol	1960	International Group
BCPL	1967	Martin Richard
B	1970	Ken Thompson
Traditional C	1972	Dennis Ritchie
K & R C	1978	Kernighan & Dennis Ritchie
ANSI C	1989	ANSI Committee
ANSI/ISO C	1990	ISO Committee
C99	1999	Standardization Committee

Features of C Language

C is the widely used language. It provides many **features** that are given below.

1. Simple
2. Machine Independent or Portable
3. Mid-level programming language
4. structured programming language
5. Rich Library
6. Memory Management
7. Fast Speed
8. Pointers

9. Recursion
10. Extensible

1) Simple
C is a simple language in the sense that it provides a structured approach (to break the problem into parts), the rich set of library functions, data types, etc.

2) Machine Independent or Portable
Unlike assembly language, c programs can be executed on different machines with some machine specific changes. Therefore, C is a machine independent language.

3) Mid-level programming language
Although, C is intended to do low-level programming. It is used to develop system applications such as kernel, driver, etc. It also supports the features of a high-level language. That is why it is known as mid-level language.

4) Structured programming language
C is a structured programming language in the sense that we can break the program into parts using functions. So, it is easy to understand and modify. Functions also provide code reusability.

5) Rich Library
C provides a lot of inbuilt functions that make the development fast.

6) Memory Management
It supports the feature of dynamic memory allocation. In C language, we can free the allocated memory at any time by calling the free() function.

7) Speed
The compilation and execution time of C language is fast since there are lesser inbuilt functions and hence the lesser overhead.

8) Pointer
C provides the feature of pointers. We can directly interact with the memory by using the pointers. We can use pointers for memory, structures, functions, array, etc.

9) Recursion
In C, we can call the function within the function. It provides code reusability for every function. Recursion enables us to use the approach of backtracking.

10) Extensible
C language is extensible because it can easily adopt new features.

First C Program

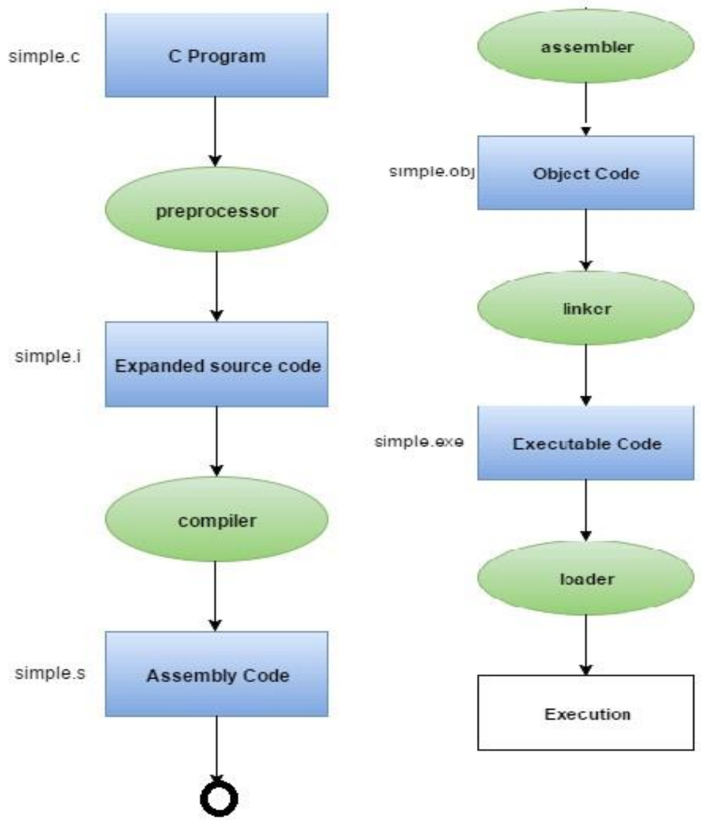
Before starting the abcd of C language, you need to learn how to write, compile and run the first c program.

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

- **#include <stdio.h>** includes the standard input output library functions. The printf() function is defined in stdio.h .
- **int main()** The main() function is the entry point of every program in c language.
- **printf()** The printf() function is used to print data on the console.
- **return 0** The return 0 statement, returns execution status to the OS. The 0 value is used for successful execution and 1 for unsuccessful execution.

Execution Flow

Let's try to understand the flow of above program by the figure given below.



1) C program (source code) is sent to preprocessor first. The preprocessor is responsible to convert preprocessor directives into their respective values. The preprocessor generates an expanded source code.

2) Expanded source code is sent to compiler which compiles the code and converts it into assembly code.

3) The assembly code is sent to assembler which assembles the code and converts it into object code. Now a simple.obj file is generated.

4) The object code is sent to linker which links it to the library such as header files. Then it is converted into executable code. A simple.exe file is generated.

5) The executable code is sent to loader which loads it into memory and then it is executed. After execution, output is sent to console.

Structure of C program

Structure of C Program	
1.Documentation Section or Comment Section	Consists of comments, some description of the program, programmer name and any other useful points.
2.Link Section	Provides instruction to the compiler to link function from the library function.
3.Definition Section	Consists of symbolic constants.
4.Global Varable declaration Section	Consists of function declaration and global variables.

5.main( ) Function
{
5.1 Local Variable section
5.2 Executable Statements
}
	Every C program must have a main() function which is the starting point of the program execution.

6.Subprograms	User defined functions.

Identifiers

Identifiers are the names given to variables, constants, functions and user-define data. These identifiers are defined against a set of rules.

Rules for an Identifier

1. An Identifier can only have alphanumeric characters (a-z, A-Z , 0-9) and underscore(_).
2. The first character of an identifier can only contain alphabet (a-z , A-Z) or underscore (_).

3. Identifiers are also case sensitive in C. For example name and Name are two different identifiers in C.
4. Keywords are not allowed to be used as Identifiers.
5. No special characters, such as semicolon, period, whitespaces, slash or comma are permitted to be used in or as Identifier.

When we declare a variable or any function, to use it we must provide a name to it, which identified it throughout the program

int myvariable=”VEMU”;

Here myvariable is the name or identifier for the variable which stores the value "VEMU" in it.

Character set

The characters are grouped into the following catagories,

1. Letters (all alphabets a to z & A to Z).
2. Digits (all digits 0 to 9).
3. Special characters, (such as colon :, semicolon ;, period ., underscore _, ampersand & etc).
4. White spaces.

printf() and scanf() Functions:

The printf() and scanf() functions are used for input and output in C language. Both functions are inbuilt library functions, defined in stdio.h (header file).

printf() function

The **printf() function** is used for output. It prints the given statement to the console.

The syntax of printf() function is given below:

Syntax: printf("format string",argument_list);

The format string can be %d (integer), %c (character), %s (string), %f (float) etc.

scanf() function

The **scanf() function** is used for input. It reads the input data from the console.

Synatx: scanf("format string",argument_list);

Program to print cube of given number

```
#include<stdio.h>
int main()
{
int number;
printf("enter a number:");
scanf("%d",&number);
printf("cube of number is:%d ",number*number*number);
return 0;
}
```

Output

```
enter a number:5
cube of number is:125
```

The scanf("%d",&number) statement reads integer number from the console and stores the given value in number variable.

The printf("cube of number is:%d ",number*number*number) statement prints the cube of number on the console.

Program to print sum of 2 numbers

```
#include<stdio.h>
int main()
{
int x=0,y=0,result=0;
printf("enter first number:");
scanf("%d",&x);
printf("enter second number:");
scanf("%d",&y);
result=x+y;
printf("sum of 2 numbers:%d ",result);
return 0;
}
```

Output

```
enter first number:9
enter second number:9
sum of 2 numbers:18
```

Variables

A **variable** is a name of the memory location. It is used to store data. Its value can be changed, and it can be reused many times.
It is a way to represent memory location through symbol so that it can be easily identified.

Syntax: type variable_list;

Ex:The example of declaring the variable

```
int a;
float b;
char c;
```

Here, a, b, c is variables. The int, float, char are the data types. We can also provide values while declaring the variables.

```c
int a=10,b=20;//declaring 2 variable of integer type
float f=20.8;
char c='A';
```

Rules for defining variables

- A variable can have alphabets, digits, and underscore.
- A variable name can start with the alphabet, and underscore only. It can't start with a digit.
- No whitespace is allowed within the variable name.
- A variable name must not be any reserved word or keyword, e.g. int, float, etc.

Valid variable names	Invalid variable names
1. int a;	**1. int** 2;
2. int _ab;	**2. int** a b;
3. int a30;	**3. int long**;

Types of Variables

1. local variable
2. global variable
3. static variable
4. automatic variable
5. external variable

Local Variable

A variable that is declared inside the function or block is called a local variable. It must be declared at the start of the block.

You must have to initialize the local variable before it is used

```c
void function1()
{
int x=10;//local variable
}
```

Global Variable / Header Variables

A variable that is declared outside the function or block is called a global variable. Any function can change the value of the global variable. It is available to all the functions.

It must be declared at the start of the block.

```c
int value=20;//global variable
void function1()
{
int x=10;//local variable
}
```

Static Variable

A variable that is declared with the static keyword is called static variable.It retains its value between multiple function calls.

```
void function1()
{
int x=10;//local variable
static int y=10;//static variable
x=x+1;
y=y+1;
printf("%d,%d",x,y);
}
```

If you call this function many times, the **local variable will print the same value** for each function call, e.g, 11,11,11 and so on. But the **static variable will print the incremented value** in each function call, e.g. 11, 12, 13 and so on.

Automatic Variable

All variables in C that are declared inside the block are automatic variables by default. We can explicitly declare an automatic variable using **auto keyword**.

```
void main()
{
int x=10;//local variable (also automatic)
auto int y=20;//automatic variable
}
```

External Variable

We can share a variable in multiple C source files by using an external variable. To declare an external variable, you need to use **extern keyword**.

```
extern int x=10;//external variable (also global)
```

```
#include "myfile.h"
#include <stdio.h>
void printValue()
{
printf("Global variable: %d", global_variable);
}
```

Data Types

A data type specifies the type of data that a variable can store such as integer, floating, character, etc.

Types	Data Types
Basic Data Type	int, char, float, double
Derived Data Type	array, pointer, structure, union
Enumeration Data Type	enum
Void Data Type	void

Basic Data Types

The basic data types are integer-based and floating-point based. C language supports both signed and unsigned literals. The memory size of the basic data types may change according to 32 or 64-bit operating system.

Let's see the basic data types. Its size is given **according to 32-bit architecture**.

Data Types	Memory Size	Range
char	1 byte	−128 to 127
signed char	1 byte	−128 to 127
unsigned char	1 byte	0 to 255
short	2 byte	−32,768 to 32,767
signed short	2 byte	−32,768 to 32,767
unsigned short	2 byte	0 to 65,535
int	2 byte	−32,768 to 32,767
signed int	2 byte	−32,768 to 32,767
unsigned int	2 byte	0 to 65,535
short int	2 byte	−32,768 to 32,767
signed short int	2 byte	−32,768 to 32,767
unsigned short int	2 byte	0 to 65,535
long int	4 byte	-2,147,483,648 to 2,147,483,647
signed long int	4 byte	-2,147,483,648 to 2,147,483,647
unsigned long int	4 byte	0 to 4,294,967,295
float	4 byte	
double	8 byte	
long double	10 byte	

Keywords

A keyword is a **reserved word**. You cannot use it as a variable name, constant name, etc. There are only 32 reserved words (keywords) in the C language.A list of 32 keywords in the c language.

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

Precedence of Operators

The precedence of operator species that which operator will be evaluated first and next. The associativity specifies the operator direction to be evaluated; it may be left to right or right to left.

int value=10+20*10;

The value variable will contain **210** because * (multiplicative operator) is evaluated before + (additive operator). The precedence and associativity of operators.

OPERATERS	OPERATION	ASSOCIATIVTY	PRECEDENCE
( )	Parentheses (function call) (see Note 1)	left-to-right	1st
[ ]	Brackets (array subscript)		
->	Member selection via pointer		
.	Member selection via object name		
+ –	Unary plus/minus	right-to-left	2nd
++ —	increment/decrement	right-to-left	
! ~	Logical negation/bitwise complement		
*****	Dereference/Pointer operater		
&	Address (of operand)		
sizeof	Determine size in bytes on this implementation		
(type)	Cast (convert value to temporary value of type)		
*** / %**	Multiplication/division/modulus	left-to-right	3rd

+ –	Addition/subtraction	left-to-right	4th
<< >>	Bitwise shift left, Bitwise shift right	left-to-right	5th
< <=	Relational less than/less than or equal to	left-to-right	6th
> >=	Relational greater than/greater than or equal to		
== !=	Relational is equal to/is not equal to	left-to-right	7th
&	Bitwise AND	left-to-right	8th
^	Bitwise exclusive OR	left-to-right	9th
|	Bitwise inclusive OR	left-to-right	10th
&&	Logical AND	left-to-right	11th
||	Logical OR	left-to-right	12th
? :	Ternary conditional	right-to-left	13th
=	Assignment	right-to-left	14th
+= -=	Addition/subtraction assignment	right-to-left	14th
*= /=	Multiplication/division assignment	right-to-left	14th
%= &=	Modulus/bitwise AND assignment	right-to-left	14th
^= |=	Bitwise exclusive/inclusive OR assignment	right-to-left	14th
<<= >>=	Bitwise shift left/right assignment	right-to-left	14th
,	Comma (separate expressions)	left-to-right	15th

Constants

Constants are also like normal variables. But, only difference is, their values cannot be modified by the program once they are defined.

- Constants refer to fixed values. They are also called as literals
- Constants may be belonging to any of the data type.

  Syntax: const data_type variable_name; (or) const data_type *variable_name;

Types of Constants:

1. Integer constants
2. Real or Floating point constants
3. Octal & Hexadecimal constants
4. Character constants
5. String constants
6. Backslash character constants

Constant type	**data type (Example)**
Integer constants	int (53, 762, -478 etc) unsigned int (5000u, 1000U etc) long int, long long int (483,647 2,147,483,680)
Real or Floating point constants	float (10.456789) doule (600.123456789)

Octal constant	int (Example: 013 /*starts with 0 */)
Hexadecimal constant	int (Example: 0x90 /*starts with 0x*/)
character constants	char (Example: ‘A’, ‘B’, ‘C’)
string constants	char (Example: “ABCD”, “Hai”)

Rules For Constructing Constant:

1. Integer Constants:

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

2. Real Constants:

- A real constant must have at least one digit
- It must have a decimal point
- It could 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 a real constant.

3. Character and String Constants:

- A character constant is a single alphabet, a single digit or a single special symbol enclosed within single quotes.
- The maximum length of a character constant is 1 character.
- String constants are enclosed within double quotes.

4. Backslash Character Constants:

- There are some characters which have special meaning in C language.
- They should be preceded by backslash symbol to make use of special function of them.
- Given below is the list of special characters and their purpose.

Backslash_character	**Meaning**
\b	Backspace
\f	Form feed

\n	New line
\r	Carriage return
\t	Horizontal tab
\”	Double quote
\’	Single quote
\\	Backslash
\v	Vertical tab
\a	Alert or bell
\?	Question mark
\N	Octal constant (N is an octal constant)
\XN	Hexadecimal constant (N – hex.dcml cnst)

How to Use Constants?

We can define constants in a C program in the following ways.

1. By “const” keyword
2. By “#define” preprocessor directive

Please note that when you try to change constant values after defining in C program, it will through error.

1. Example Program Using Const Keyword :

```
#include <stdio.h>
void main()
{
const int height = 100;       /*int constant*/
const float number = 3.14; /*Real constant*/
const char letter = 'A';       /*char constant*/
const char letter_sequence[10] = "ABC";    /*string constant*/
const char backslash_char = '\?';   /*special char cnst*/
printf("value of height :%d \n", height );
printf("value of number : %f \n", number );
printf("value of letter : %c \n", letter );
printf("value of letter_sequence : %s \n", letter_sequence);
printf("value of backslash_char : %c \n", backslash_char);
}
```

Output:

```
Value of height : 100
value of number : 3.140000
value of letter : A
value of letter_sequence : ABC
value of backslash_char : ?
```

3. **Example Program Using #Define Preprocessor Directive**

```
#include <stdio.h>
#define height 100
#define number 3.14
#define letter 'A'
#define letter_sequence "ABC"
#define backslash_char '\?'
void main()
{
printf("value of height : %d \n", height );
printf("value of number : %f \n", number );
printf("value of letter : %c \n", letter );
printf("value of letter_sequence : %s \n",letter_sequence);
printf("value of backslash_char : %c \n",backslash_char);
}
```

Output:

```
value of height : 100
value of number : 3.140000
value of letter : A
value of letter_sequence : ABC
value of backslash_char : ?
```

Tokens and keywords

Tokens, Identifiers and Keywords are the basics in a C program.

1. C Tokens:

- C tokens are the basic buildings blocks in C language which are constructed together to write a C program.
- Each and every smallest individual unit in a C program is known as C tokens.

C tokens are of six types. They are,

1.	Keywords	(eg: int, while),
2.	Identifiers	(eg: main, total),
3.	Constants	(eg: 10, 20),
4.	Strings	(eg: "total", "hello"),
5.	Special symbols	(eg: (), {}),
6.	Operators	(eg: +, /,-,*)

C Tokens Example Program:

```
int main()
{
  int x, y, total;
  x = 10, y = 20;
  total = x + y;
  printf ("Total = %d \n", total);
}
```

where,

- main – identifier
- {,}, (,) – delimiter
- int – keyword
- x, y, total – identifier
- main, {, }, (,), int, x, y, total – tokens

Do you know how to use C token in real time application programs? We have given simple real time application programs where C token is used. You can refer the below C programs to know how to use C token in real time program.

2. **Identifiers in C Language:**

- Each program elements in a C program are given a name called identifiers.
- Names given to identify Variables, functions and arrays are examples for identifiers. eg. x is a name given to integer variable in above program.

Rules for Constructing Identifier Name:

1. First character should be an alphabet or underscore.
2. Succeeding characters might be digits or letter.
3. Punctuation and special characters aren't allowed except underscore.
4. Identifiers should not be keywords.

Operators and Expressions

- ➢ The symbols which are used to perform logical and mathematical operations are called C operators.
- ➢ These operators join individual constants and variables to form expressions.
- ➢ Operators, functions, constants and variables are combined together to form expressions.
- ➢ Consider the expression A + B * 5. where, +, * are operators, A, B are variables, 5 is constant and A + B * 5 is an expression.

Types of C Operators

1. Arithmetic operators
2. Assignment operators
3. Relational operators
4. Logical operators
5. Bit wise operators
6. Conditional operators (ternary operators)
7. Increment/decrement operators
8. Special operators

Arithmetic Operators:

Arithmetic operators are used to perform mathematical calculations like addition, subtraction, multiplication, division and modulus.

Arithmetic Operators/Operation	**Example**
+ (Addition)	A+B
– (Subtraction)	A-B
* (multiplication)	A*B
/ (Division)	A/B
% (Modulus)	A%B

Example Program

```
#include <stdio.h>
 Void  main()
{
   int a=40,b=20, add,sub,mul,div,mod;
   add = a+b;
   sub = a-b;
   mul = a*b;
   div = a/b;
   mod = a%b;
   printf("Addition of a, b is : %d\n", add);
   printf("Subtraction of a, b is : %d\n", sub);
   printf("Multiplication of a, b is : %d\n", mul);
   printf("Division of a, b is : %d\n", div);
   printf("Modulus of a, b is : %d\n", mod);
}
```

Output

```
            Addition of a, b is : 60
            Subtraction of a, b is : 20
            Multiplication of a, b is : 800
            Division of a, b is : 2
            Modulus of a, b is : 0
```

Assignment Operators:

- The values for the variables are assigned using assignment operators.
- For example, if the value “10” is to be assigned for the variable “sum”, it can be assigned as “sum = 10;”
- There are 2 categories of assignment operators in C language. They are,
  1. Simple assignment operator (Example: =)
  2. Compound assignment operators (Example: +=, -=, *=, /=, %=, &=, ^=)

Operators	Example/Description
=	sum = 10; 10 is assigned to variable sum
+=	sum += 10; This is same as sum = sum + 10

-=	sum -= 10; This is same as sum = sum – 10
*=	sum *= 10; This is same as sum = sum * 10
/=	sum /= 10; This is same as sum = sum / 10
%=	sum %= 10; This is same as sum = sum % 10
&=	sum&=10; This is same as sum = sum & 10
^=	sum ^= 10; This is same as sum = sum ^ 10

Example Program for Assignment Operators:

- In this program, values from 0 – 9 are summed up and total “45” is displayed as output.

Assignment operators such as “=” and “+=” are used in this program to assign the values and to sum up the values.

```
# include <stdio.h>
 int main()
{
int Total=0,i;
1 for(i=0;i<10;i++)
{
Total+=i; // This is same as Total = Toatal+i
}
printf("Total = %d", Total);
}
```

OUTPUT:

```
Total = 45
```

Relational Operators:

Relational operators are used to find the relation between two variables. i.e. to compare the values of two variables in a C program.

Operators	Example/Description
>	x > y (x is greater than y)

<	x < y (x is less than y)
>=	x >= y (x is greater than or equal to y)
<=	x <= y (x is less than or equal to y)
==	x == y (x is equal to y)
!=	x != y (x is not equal to y)

Example Program For Relational Operators :

- In this program, relational operator (= =) is used to compare 2 values whether they are equal are not.
- If both values are equal, output is displayed as ” values are equal”. Else, output is displayed as “values are not equal”.

Note: double equal sign (= =) should be used to compare 2 values. We should not single equal sign (=).

```
#include <stdio.h>
 int main()
{
   int m=40,n=20;
   if (m == n)
   {
       printf("m and n are equal");
   }
   else
   {
       printf("m and n are not equal");
   }
}
```

Output:

```
m and n are not equal
```

Logical Operators:

- These operators are used to perform logical operations on the given expressions.
- There are 3 logical operators in C language. They are, logical AND (&&), logical OR (||) and logical NOT (!).

Operators	Example/Description
&& (logical AND)	(x>5)&&(y<5) It returns true when both conditions are true
|| (logical OR)	(x>=10)||(y>=10) It returns true when at-least one of the condition is true
! (logical NOT)	!((x>5)&&(y<5)) It reverses the state of the operand "((x>5) && (y<5))" If "((x>5) && (y<5))" is true, logical NOT operator makes it false

Example Program For Logical Operators:

```
 #include <stdio.h>
  int main()
 {
   int m=40,n=20;
   int o=20,p=30;
   if (m>n && m !=0)
   {
     printf("&& Operator : Both conditions are true\n");
   }
   if (o>p || p!=20)
   {
     printf("|| Operator : Only one condition is true\n");
   }
   if (!(m>n && m !=0))
   {
     printf("! Operator : Both conditions are true\n");
   }
   else
   {
     printf("! Operator : Both conditions are true. " \
     "But, status is inverted as false\n");
   }
 }
```

Output:

```
&& Operator : Both conditions are true
```

|| Operator : Only one condition is true
! Operator : Both conditions are true. But, status is inverted as false

- In this program, operators (&&, || and !) are used to perform logical operations on the given expressions.

- **&& operator** – "if clause" becomes true only when both conditions (m>n and m! =0) is true. Else, it becomes false.

- **|| Operator** – "if clause" becomes true when any one of the condition (o>p || p!=20) is true. It becomes false when none of the condition is true.
- **! Operator** – It is used to reverses the state of the operand.
- If the conditions (m>n && m! =0) is true, true (1) is returned. This value is inverted by "!" operator.
- So, "! (m>n and m! =0)" returns false (0).

Bit Wise Operators:

- These operators are used to perform bit operations. Decimal values are converted into binary values which are the sequence of bits and bit wise operators work on these bits.

- Bit wise operators in C language are & (bitwise AND), | (bitwise OR), ~ (bitwise OR), ^ (XOR), << (left shift) and >> (right shift).

Truth Table for Bit Wise Operation & Bit Wise Operators:

x	y	x|y	x&y	x^y
0	0	0	0	0
0	1	1	0	1
1	0	1	0	1
1	1	1	1	0

Below Are the Bit-Wise Operators And.

1. & – Bitwise AND
2. | – Bitwise OR
3. ~ – Bitwise NOT
4. ^ – XOR
5. << – Left Shift
6. >> – Right Shift

Consider x=40 and y=80. Binary form of these values are given below.

x = 00101000
y= 01010000

All bit wise operations for x and y are given below.

1. x&y = 00000000 (binary) = 0 (decimal)
2. x|y = 01111000 (binary) = 120 (decimal)
3. ~x = 1111111111111111111111111 11111111111111111111111111111111010111 = -41 (decimal)
4. x^y = 01111000 (binary) = 120 (decimal)
5. x << 1 = 01010000 (binary) = 80 (decimal)
6. x >> 1 = 00010100 (binary) = 20 (decimal)

Note:

- **Bit wise NOT :** Value of 40 in binary is 0000000000000000000000000000000 00000000000000000010100000000000. So, all 0's are converted into 1's in bit wise NOT operation.

- **Bit wise left shift and right shift :** In left shift operation "x << 1 ", 1 means that the bits will be left shifted by one place. If we use it as "x << 2 ", then, it means that the bits will be left shifted by 2 places.

Example Program For Bit Wise Operators:

In this example program, bit wise operations are performed as shown above and output is displayed in decimal format.

```
#include <stdio.h>
 int main()
{
   int m = 40,n = 80,AND_opr,OR_opr,XOR_opr,NOT_opr ;
   AND_opr = (m&n);
   OR_opr = (m|n);
   NOT_opr = (~m);
   XOR_opr = (m^n);
   printf("AND_opr value = %d\n",AND_opr );
   printf("OR_opr value = %d\n",OR_opr );
   printf("NOT_opr value = %d\n",NOT_opr );
   printf("XOR_opr value = %d\n",XOR_opr );
   printf("left_shift value = %d\n", m << 1);
```

```
    printf("right_shift value = %d\n", m >> 1);
  }
```

Output:

```
AND_opr value = 0
OR_opr value = 120
NOT_opr value = -41
XOR_opr value = 120
left_shift value = 80
right_shift value = 20
```

Conditional or Ternary Operators:

- Conditional operators return one value if condition is true and returns another value is condition is false.
- This operator is also called as ternary operator.

Syntax : (Condition? true_value: false_value);

Example: (A > 100? 0: 1);

- In above example, if A is greater than 100, 0 is returned else 1 is returned. This is equal to if else conditional statements.

Example Program For Conditional/Ternary Operators:

```
 #include <stdio.h>
  int main()
 {
    int x=1, y ;
    y = ( x ==1 ? 2 : 0 ) ;
    printf("x value is %d\n", x);
    printf("y value is %d", y);
 }
```

Output:

```
x value is 1
y value is 2
```

Increment/decrement Operators

- Increment operators are used to increase the value of the variable by one and decrement operators are used to decrease the value of the variable by one in C programs.

- **Syntax:**
  Increment operator: ++var_name; (or) var_name++;
  Decrement operator: – -var_name; (or) var_name – -;

- **Example:**

  Increment operator : ++ i ; i ++ ;
  Decrement operator : – – i ; i – – ;

Example Program For Increment Operators:

In this program, value of "i" is incremented one by one from 1 up to 9 using "i++" operator and output is displayed as "1 2 3 4 5 6 7 8 9".

```
 #include <stdio.h>
int main()
{
    int i=1;
    while(i<10)
    {
       printf("%d ",i);
       i++;
    }
}
```

Output:

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

Example Program for Decrement Operators:

In this program, value of "I" is decremented one by one from 20 up to 11 using "i–" operator and output is displayed as "20 19 18 17 16 15 14 13 12 11".

```
 #include <stdio.h>
int main()
{
   int i=20;
   while(i>10)
   {
      printf("%d ",i);
      i--;
   }
}
```

Output:

```
20 19 18 17 16 15 14 13 12 11
```

Difference between Pre/Post Increment & Decrement Operators:

Below table will explain the difference between pre/post increment and decrement operators

Operator	Operator/Description
Pre increment operator (++i)	value of i is incremented before assigning it to the variable i
Post increment operator (i++)	value of i is incremented after assigning it to the variable i
Pre decrement operator (–i)	value of i is decremented before assigning it to the variable i
Post decrement operator (i–)	value of i is decremented after assigning it to variable i

Example Program For Pre – Increment Operators:

```
#include <stdio.h>
int main()
{
          int i=0;
          while(++i < 5 )
          {
              printf("%d ",i);
          }
          return 0;
}
```

Output:

```
1 2 3 4
```

- Step 1 : In above program, value of “i” is incremented from 0 to 1 using pre-increment operator.

- Step 2 : This incremented value “1” is compared with 5 in while expression.
- Step 3 : Then, this incremented value “1” is assigned to the variable “i”.
- Above 3 steps are continued until while expression becomes false and output is displayed as “1 2 3 4”.

Example Program for Post – Increment Operators:

```
#include <stdio.h>
int main()
{
        int i=0;
        while(i++ < 5 )
        {
            printf("%d ",i);
        }
        return 0;
}
```

Output:

```
1 2 3 4 5
```

- Step 1 : In this program, value of i “0” is compared with 5 in while expression.
- Step 2 : Then, value of “i” is incremented from 0 to 1 using post-increment operator.
- Step 3 : Then, this incremented value “1” is assigned to the variable “i”.
- Above 3 steps are continued until while expression becomes false and output is displayed as “1 2 3 4 5”.

Example Program For Pre – Decrement Operators:

```
#include <stdio.h>
int main()
{
        int i=10;
        while(--i > 5 )
        {
            printf("%d ",i);
        }
        return 0;
}
```

Output:

```
9 8 7 6
```

- Step 1 : In above program, value of “i” is decremented from 10 to 9 using pre-decrement operator.
- Step 2 : This decremented value “9” is compared with 5 in while expression.
- Step 3 : Then, this decremented value “9” is assigned to the variable “i”.
- Above 3 steps are continued until while expression becomes false and output is displayed as “9 8 7 6”.

Example Program For Post – Decrement Operators:

```
#include <stdio.h>
int main()
{
        int i=10;
        while(i-- > 5 )
        {
            printf("%d ",i);
        }
        return 0;
}
```

Output:

```
9 8 7 6 5
```

- Step 1 : In this program, value of i “10” is compared with 5 in while expression.
- Step 2 : Then, value of “i” is decremented from 10 to 9 using post-decrement operator.
- Step 3 : Then, this decremented value “9” is assigned to the variable “i”.
- Above 3 steps are continued until while expression becomes false and output is displayed as “9 8 7 6 5”.

Special Operators:

Below are some of the special operators that the C programming language offers.

Operators	**Description**
&	This is used to get the address of the variable. Example : &a will give address of a.

*	This is used as pointer to a variable. Example : * a where, * is pointer to the variable a.
Sizeof ()	This gives the size of the variable. Example : size of (char) will give us 1.

Example Program for & and * Operators:

In this program, “&” symbol is used to get the address of the variable and “*” symbol is used to get the value of the variable that the pointer is pointing to. Please refer **C – pointer** topic to know more about pointers.

```
#include <stdio.h>
int main()
{
int *ptr, q;
q = 50;
/* address of q is assigned to ptr */
ptr = &q;
/* display q's value using ptr variable */
printf("%d", *ptr);
return 0;
}
```

Output:

```
50
```

Example Program For Sizeof() Operator :

sizeof() operator is used to find the memory space allocated for each C data types.

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

int main()
{
int a;
char b;
float c;
double d;
printf("Storage size for int data type:%d \n",sizeof(a));
printf("Storage size for char data type:%d \n",sizeof(b));
printf("Storage size for float data type:%d \n",sizeof(c));
printf("Storage size for double data type:%d\n",sizeof(d));
return 0;
}
```

Output:

```
Storage size for int data type:4
Storage size for char data type:1
Storage size for float data type:4
Storage size for double data type:8
```

Types of C Type Qualifiers:

There are two types of qualifiers available in C language. They are,

1. const
2. volatile

1. Const Keyword:

- Constants are also like normal variables. But, only difference is, their values can't be modified by the program once they are defined.
- They refer to fixed values. They are also called as literals.
- They may be belonging to any of the data type.
- **Syntax:**
  const data_type variable_name; (or) const data_type *variable_name;

2. Volatile Keyword:

- When a variable is defined as volatile, the program may not change the value of the variable explicitly.
- But, these variable values might keep on changing without any explicit assignment by the program. These types of qualifiers are called volatile.
- For example, if global variable's address is passed to clock routine of the operating system to store the system time, the value in this address keep on changing without any assignment by the program. These variables are named as volatile variable.
- **Syntax:**
  volatile data_type variable_name; (or) volatile data_type *variable_name;

Storage Class Specifies / Scope variables

Storage class specifies in C language tells the compiler where to store a variable, how to store the variable, what is the initial value of the variable and life time of the variable.

Syntax:

storage_specifier data_type variable _name;

Types Of Storage Class Specifiers:

There are 4 storage class specifiers available in C language. They are,

1. auto
2. extern
3. static
4. register

Storage Specifies	Description
auto	Storage place: CPU Memory Initial/default value: Garbage value Scope: local Life: Within the function only.
extern	Storage place: CPU memory Initial/default value: Zero Scope: Global Life: Till the end of the main program. Variable definition might be anywhere in the program.
static	Storage place: CPU memory Initial/default value: Zero Scope: local Life: Retains the value of the variable between different function calls.
register	Storage place: Register memory Initial/default value: Garbage value Scope: local Life: Within the function only.

NOTE:

- For faster access of a variable, it is better to go for register specifiers rather than auto specifies.
- Because, register variables are stored in register memory whereas auto variables are stored in main CPU memory.
- Only few variables can be stored in register memory. So, we can use variables as register that are used very often in a C program.

1. Example Program For Auto Variable :

The scope of this auto variable is within the function only. It is equivalent to local variable. All local variables are auto variables by default.

```
#include<stdio.h>
void increment(void);
int main()
{
   increment();
   increment();
   increment();
   increment();
   return 0;
}
 void increment(void)
{
   auto int i = 0 ;
   printf ( "%d ", i ) ;
   i++;
}
```

Output:

```
0 0 0 0
```

2. Example Program For Static Variable :

Static variables retain the value of the variable between different function calls.

```
//C static example
#include<stdio.h>
void increment(void);
int main()
{
increment();
increment();
increment();
increment();
return 0;
}
void increment(void)
{
static int i = 0 ;
printf ( "%d ", i ) ;
i++;
}
```

Output:

```
0 1 2 3
```

3. Example Program For Extern Variable :

The scope of this extern variable is throughout the main program. It is equivalent to global variable. Definition for extern variable might be anywhere in the C program.

```
#include<stdio.h>
 int x = 10 ;
int main( )
{
extern int y;
printf("The value of x is %d \n",x);
printf("The value of y is %d",y);
return 0;
}
int y=50;
```

Output:

```
The value of x is 10
The value of y is 50
```

4. Example Program For Register Variable:

- Register variables are also local variables, but stored in register memory. Whereas, auto variables are stored in main CPU memory.
- Register variables will be accessed very faster than the normal variables since they are stored in register memory rather than main memory.

But, only limited variables can be used as register since register size is very low. (16 bits, 32 bits or 64 bits)

```
#include <stdio.h>
int main()
{
  register int i;
  int arr[5];// declaring array
  arr[0] = 10;// Initializing array
  arr[1] = 20;
  arr[2] = 30;
  arr[3] = 40;
  arr[4] = 50;
  for (i=0;i<5;i++)
  {
    // Accessing each variable
    printf("value of arr[%d] is %d \n", i, arr[i]);
```

```
   }
   return 0;
 }
```

Output:

```
value of arr[0] is 10
value of arr[1] is 20
value of arr[2] is 30
value of arr[3] is 40
value of arr[4] is 50
```

Control Statements

A **statement** is a part of your program that can be executed. That is, a statement specifies an action. Statements generally contain expressions and end with a semicolon. Statements that are written individually are called **Single/Simple statements**. Statements that are written as a block are called **Block/Compound statements**. A block begins with an open brace {and ends with a closing brace}.

C categorizes statements into these groups:

- Selection / Branch / Decision making / Conditional statements
- Loop/ Iteration/ Repetitive statements
- Jump / Control transfer statements
- Expression Statements
- Block statements/ compound statements

Selection Statements:

A **selection statement** checks the given condition and decides the execution of statements after the success or failure of the condition. C supports two selection control statements **if** and **switch**.

1. If statement
2. Switch Statement

1. If statement:

The **if** statement is a decision making statement that allows the computer to evaluate an expression (condition) first and depending on the result of the condition i.e. true or false, it transfers the control to a particular statement.

The if statement has the following forms

- a) Simple if Statement
- b) if...else Statement
- c) Nested if Statement

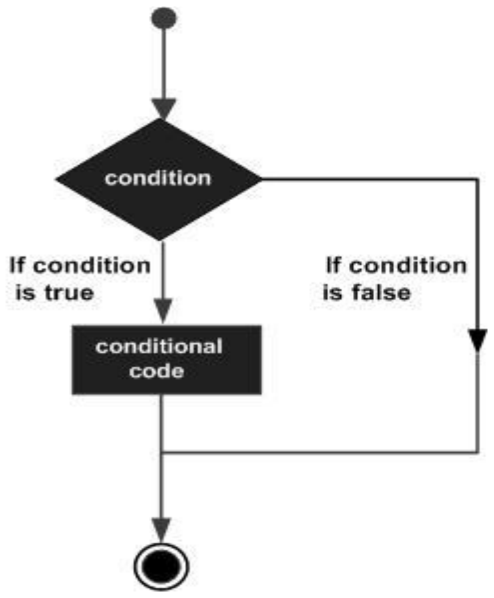
- d) If...else Ladder Statement

Simple if Statement

The simple if statement contains only one condition, if the condition is true, it will execute the statements that are present between opening and closing braces. Otherwise it will not execute those statements.

Syntax:	**Example:/* to check whether a num is Less than 100 or not */**

if(condition**)**
Single-statement
OR
if(condition**)**
{
Statement-block
}
	#include<stdio.h>
	#include<conio.h>
	void main()
	{
	int a;
	clrscr();
	printf("enter an integer ");
	scanf("%d",&a);
	if(a<=10)
	printf("%d is less than 100",a);
	getch();
	}

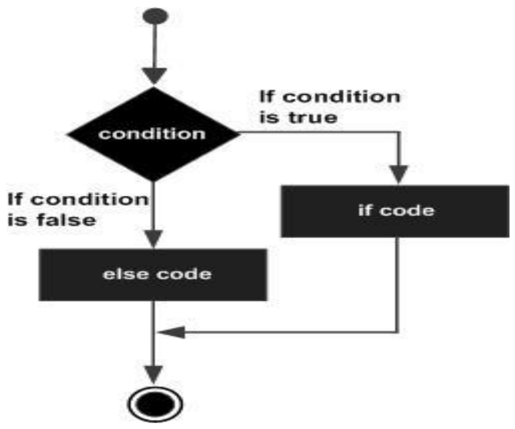


If...else Statement

This statement is used to define two blocks of statements in order to execute only one block. If the condition is true, the block of **if** is executed; otherwise, the block of **else** is executed.

Syntax:	**Example:**

if (condition)
{
True Block Statements;
------------- 	
}
else
{
False Block Statements;
------------ 	
}
	to check whether the given num is a positive number or a negative number
	#include<stdio.h>
	#include<conio.h>
	void main()
	{
	int n;
	clrscr();
	printf("enter any number ");
	scanf("%d",&n);
	if(n>=0)
	{
	printf("\n %d is a positive number",n);
	}
	else
	{
	printf("\n %d is a negative number",n);
	}
	getch();
	}



Nested if –else Statement

When an If statement is placed in another if statement or in else statement, then it is called nested if statement. The nested **if-else** is used when a series of decisions are involved. In a nested **if-else,** an else statement always refers to the nearest **if** statement which is within the same block as the **else** and that is not already associated with an **else**.

Syntax:	**Example: Write a program to finding greatest among three numbers**

if (condition)
{
if(condition)
{
True Block Statements;
------------- 	
}
}
else
{
False Block Statements;
------------ 	
}
	#include<stdio.h> #include<conio.h>
	void main()
	{
	int a,b;
	clrscr();
	printf("enter any two number ");
	scanf("%d%d",&a,&b);
	if(a>b)
	{
	if(a>c)
	printf(" %d is greatest",a);
	else
	printf(" %d is greatest",c);
	}
	else
	{
	if(b>c)
	printf(" %d is greatest",b);
	else
	printf(" %d is greatest",c);
	}
	getch();
	}

If..Else Ladder Statement

The **if-else** ladder is used when multipath (multiple) decisions are involved. A multipath decision is a chain of **if-else**s in which the statement associated with each **else** is an **if-**statement.

Syntax:	**Example:**

if(condtion1)
{
statements1;
}
else if(condition2)
{
statements2;
}
else if(condition3)
{
statements3;
}
|
	Write a program to test whether the given number is a single digit or a double digit or a trible digit or more than three digits.
	
	#include<stdio.h>
	#include<conio.h>
	void main()
	{
	int n;
	clrscr();
	printf("enter any number");
	scanf("%d",&n);
	if(n>=0 && n<=9)
	printf("\n %d is a single digit number",n);

|
|
else if(conditionN)
{
statementsN;
}
else
{
statements;
}
	else if(n>=10 && n<=99)
	printf("\n %d is a double digit number",n);
	else if(n>=100 && n<=999)
	printf("\n %d is a trible digit number",n);
	else
	printf("\n %d is more than three digits",n);
	getch();
	}

2. Switch Statement

C has a built-in multiple-branch selection statement, called **switch**, which successively tests the value of an expression against a list of integer or character constants. When a match is found, the statements associated with that constant are executed.

Syntax:	**Example:**

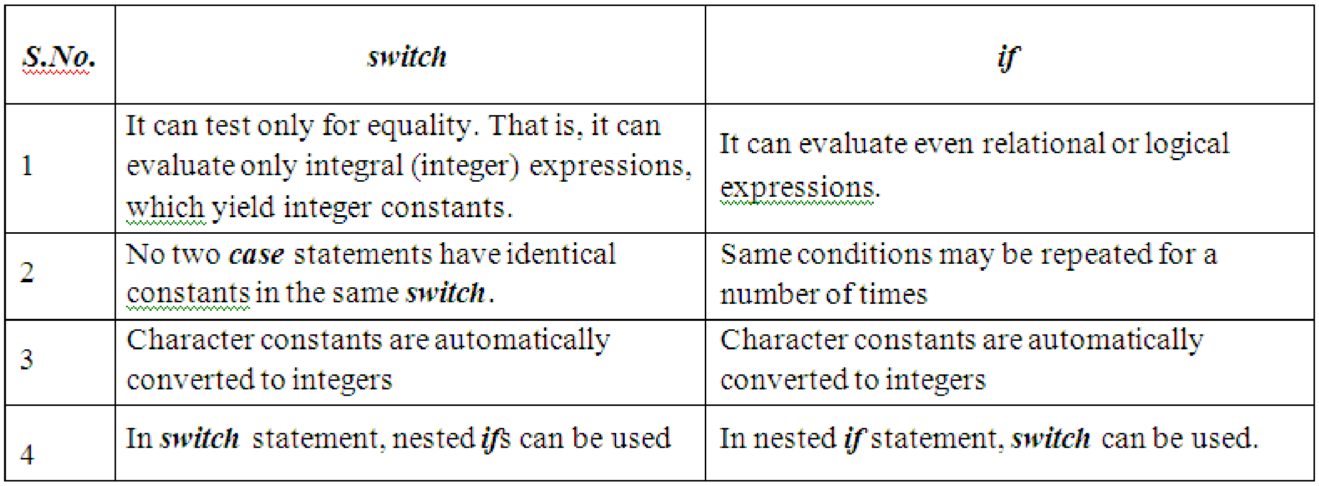
switch(expression)
{
case value1:
statements1;
break;
case value2:
statements2;
break;
|
|
|
case valueN:
statementsN;
break;
default:
statements;
}
	Write a program to check whether the given character is a vowel or not using switch case
	
	#include<stdio.h>
	#include<conio.h>
	void main()
	{
	char ch;
	clrscr();
	printf("enter any single character:");
	scanf("%c",&ch);
	switch(ch)
	{
	case 'a':
	printf("%c is a vowel",ch);
	break;
	case 'e':
	printf("%c is a vowel",ch);
	break;
	case 'i':
	printf("%c is a vowel",ch);
	break;

	case 'o':
	printf("%c is a vowel ",ch);
	break;
	case 'u':
	printf("%c is a vowel",ch);
	break;
	default:
	printf("\n %c is not a vowel",ch);
	}
	getch();
	}

The **expression** must evaluate to an integer type. Thus, you can use character or integer values, but floating point expressions, for example, are not allowed. The value of **expression** is tested against the values, one after another, of the constants specified in the **case** statements. When a match is found, the statement sequence associated with that **case** is executed until the break statement or the end of the **switch** statement is reached.

The **default** statement is executed if no matches are found. The **default** is optional, and if it is not present, no action takes place if all matches fail. Technically, the **break** statements inside the switch statement are optional. They terminate the statement sequence associated with each constant. If the break statement is omitted, execution will continue on into the next case's statements until either a break or the end of the switch is reached.

<u>Difference between if & Switch</u>



S.No.	***switch***	***if***
1	It can test only for equality. That is, it can evaluate only integral (integer) expressions, which yield integer constants.	It can evaluate even relational or logical expressions.
2	No two ***case*** statements have identical constants in the same ***switch***.	Same conditions may be repeated for a number of times
3	Character constants are automatically converted to integers	Character constants are automatically converted to integers
4	In ***switch*** statement, nested ***if***s can be used	In nested ***if*** statement, ***switch*** can be used.

Loop/ Iteration Statements

1. **While loop**
2. **do while loop**
3. **For loop**

In C, and all other modern programming languages, iteration statements (also called loops) allow a set of instructions to be repeatedly executed until a certain condition is reached. Based on position of loop, loop statements are classified into two types:

1. **entry-controlled loop (pre-test loop)- while, for**
2. **exit-controlled loop (post-test loop)- do while**

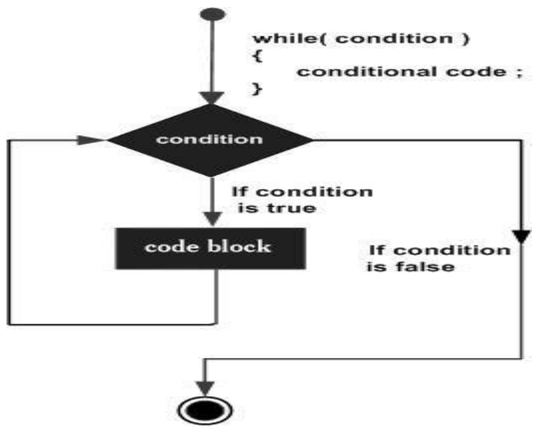
1. While loop

While is pre tested/ entry controlled loop statement i.e first condition is checked & body of loop is executed.

Syntax:	**Example:Write a program to print first 25 natural numbers**

initialization;
while(test condition)
{
Statements;
Increment / Decrement operation;
}
	#include<stdio.h> #include<conio.h>
	void main()
	{
	int n;
	clrscr();
	n=1;
	while(n<=25)
	{
	printf("%3d",n);
	n=n+1;
	}
	getch();
	}

- The **initialization** is an assignment statement that is used to set the loop control variable.
- The **condition** is a relational expression that determines when the loop exits.
- The **increment/decrement** defines how the loop control variable changes each time the loop is repeated.

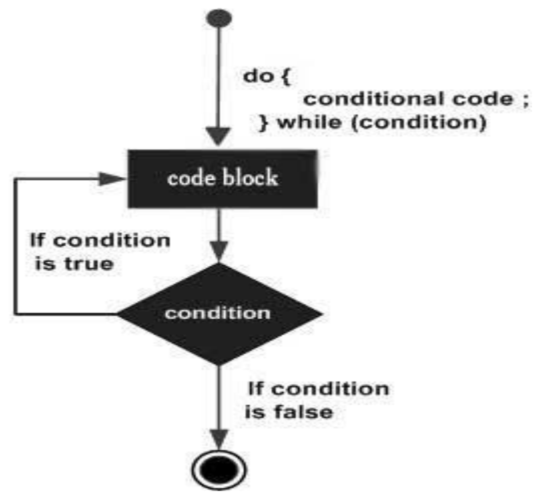


2. do while loop

Do While is post tested/ exit controlled loop statement i.e first body of loop is executed & finally condition is checked. Even though condition is false, it will execute the body of loop statements at least once.

Syntax:	**Example:**

Initialization;
do
{
Statements;
Increment/Decrement operation;
} while(test condition) ;
	Write a program to print first 25 natural numbers
	
	##include<stdio.h>
	#include<conio.h>
	void main()
	{
	int n;
	clrscr();
	n=1;
	do
	{
	printf("%3d",n);
	n=n+1;
	} while(n<=25);
	getch();
	}



Difference between while/for & do while:

- In while, if the condition is false, it will never execute the loop statements.
- But in do-while, even though condition is false, it will execute the loop statements at least once.

While Example:	**Example:**

Write a program to illustrate while

```
##include<stdio.h>
#include<conio.h>
void main()
{
int n;
clrscr( );
n=1;
while(n>10)
{
printf(“%d\n”,n);
n=n+1;
}
getch( );
}
```

output: No output because condition is false

Write a program to illustrate do while

```
##include<stdio.h>
#include<conio.h>
void main()
{
int n;
clrscr( );
n=1;
do
{
printf("%d",n);
n=n+1;
} while(n>10);
getch();
}
```

output: it prints 1 even though condition is false

3. for loop

For is pre tested/ entry controlled loop statement i.e first condition is checked & body of loop is executed.

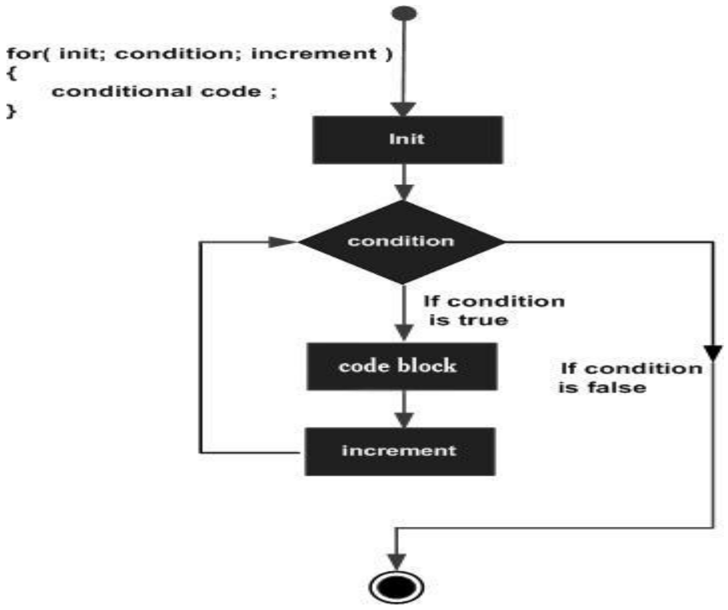
Syntax:

```
For (initialization; condition; increment /
decrement operation ;)
{
list of statements
}
```

Example:

Write a program to print first 10 numbers

```
#include<stdio.h>
#include<conio.h>
void main()
{
int i;
clrscr();
for(i=1; i<=10; i++)
{
printf(“%d\t”,i);
}
getch();
}
```



However, if condition section is omitted, the for loop becomes an endless loop, which is called an infinite loop. When the condition is absent, it is assumed to be true. The for statement may have an initialization and increment/decrement sections. But C programmers more commonly use for (;;)

Jump Statements/ Control Transfer Statements

1. Return Statement
2. Break Statement
3. Continue Statement
4. Goto Statement

C has four statements that perform either a conditional or unconditional branch: **return, goto, break, and continue.** Of these, we can use return and goto anywhere inside a function and the break and continue statements in conjunction with any of the loop statements. We can also use break with switch.

1. Return Statement

A return may or may not have a value associated with it. A **return with a value** can be used only in a function with a non-void return type. In this case, the value associated with return becomes the return value of the function. A return **without a value** is used to return (exit) from a void function. The general form of the **return** statement is **return; (OR) return** expression;

The expression may be a constant, variable, or an expression. The expression is present only in non-**void** function. In this case, the value of expression will become the return value of the function. The expression is not present in **void** function.

2. Break Statement

The break statement has two uses

a) To terminate a case in the switch statement.

b) To force immediate termination of a loop, bypassing the normal loop conditional test. The general form of the **break** statement is break keyword followed by semicolon

break;

3. Continue Statement

During the loop operations, it may be necessary to skip a part of the body of the loop under certain conditions. Like the break statement, C supports another similar statement called the **continue** statement.

However, unlike the break which causes the loop to be terminated, the continue causes the loop to be continued with the next iteration after skipping any statements in between. In **while** and **do-while** loops, **continue** causes the control to go to directly to the test condition and then to continue the iteration process. In the case of **for** loop, **continue** causes the control to go to the increment/decrement section of the loop and then to test condition. The general form of the **continue** statement is continue keyword followed by semicolon

continue;

Example:

Write a program to illustrate break

```
#include<stdio.h>
#include<conio.h>
Void main ()
{
int i;
clrscr();
for(i=1; i<=10; i++)
{
if(i==6)
break;
printf("%d\t",i);
}
getch();
}
```

Output: 1 2 3 4 5

Example:

Write a program to illustrate continue

```
#include<stdio.h>
#include<conio.h>
Void main ()
{
int i;
clrscr();
for(i=1; i<=10; i++)
{
if(i==6)
continue;
printf("%d\t",i);
}
getch();
}
```

Output: 1 2 3 4 5 7 8 9 10

4. goto Statement

The goto statement is used to branch from one point (statement) to another in the program.

The **goto** statement requires a label in order to identify the place where the branch is to be made. A **label** is a valid identifier followed by a colon. The label is placed immediately before the statement where the control is to be transferred. Furthermore, the label must be in the same function as the **goto** that uses it – we can't jump between functions.

The general form of goto is

label: --- ---- goto label;	goto label; ---- ---- label:

The **goto** breaks the normal sequential execution of the program.
If the label is before the **goto** statement, a loop will be formed and some statements will be executed repeatedly. Such a jump is known as a **backward jump**. If the label is placed after the **goto** statement, some statements will be skipped and the jump is known as a **forward jump**.

Example:

Write a program to illustrate goto statement

```
#include<stdio.h>
#include<conio.h>
void main()
{
int i;
clrscr();
for(i=1; i<=10; i++)
{
if(i==6)
goto xyz;
printf("%d\t",i);
 }
xyz: printf("\nthankyou\n");
getch();
}
```

Output: 1 2 3 4 5

Function s and Program Structures

C functions are basic building blocks in a program. All C programs are written using functions to improve re-usability, understandability and to keep track on them. You can learn below concepts of C functions in this section in detail.

Definition:

A large C program is divided into basic building blocks called C function. C function contains set of instructions enclosed by "{ }" which performs specific operation in a C program. Actually, Collection of these functions creates a C program.

USES OF C FUNCTIONS:

- C functions are used to avoid rewriting same logic/code again and again in a program.
- There is no limit in calling C functions to make use of same functionality wherever required.

- We can call functions any number of times in a program and from any place in a program.
- A large C program can easily be tracked when it is divided into functions.
- The core concept of C functions are, re-usability, dividing a big task into small pieces to achieve the functionality and to improve understandability of very large C programs.

There are 3 aspects in each function. They are

- **Function declaration or prototype:** This informs compiler about the function name, function parameters and return value's data type.
- **Function call :** This calls the actual function
- **Function definition:** This contains all the statements to be executed.

C functions aspects	syntax
function definition	Return_type function_name (arguments list) { Body of function; }
function call	function_name (arguments list);
function declaration	return_type function_name (argument list);

Simple Example Program for C Function:

- As you know, functions should be declared and defined before calling in a program.
- In the below program, function "square" is called from main function.
- The value of "m" is passed as argument to the function "square". This value is multiplied by itself in this function and multiplied value "p" is returned to main function from function "square".

```
#include<stdio.h>
float square ( float x );        // function prototype, also called function declaration
 int main( )      // main function, program starts from here
 {
   float m, n ;
   printf ( "\nEnter some number for finding square \n");
   scanf ( "%f", &m ) ;
    n = square ( m ) ;   // function call
   printf ( "\nSquare of the given number %f is %f",m,n );
}
 float square ( float x )             // function definition
{
   float p ;
   p = x * x ;
   return ( p ) ;
}
```

Output:

```
Enter some number for finding square
2
Square of the given number 2.000000 is 4.000000
```

How to Call C Functions In A Program?

There are two ways that a C function can be called from a program. They are

1. Call by value
2. Call by reference

Call By Value:

- In call by value method, the value of the variable is passed to the function as parameter.
- The value of the actual parameter cannot be modified by formal parameter.
- Different Memory is allocated for both actual and formal parameters. Because, value of actual parameter is copied to formal parameter.

Note:

- Actual parameter – This is the argument which is used in function call.
- Formal parameter – This is the argument which is used in function definition

Example Program For C Function (Using Call By Value):

- In this program, the values of the variables “m” and “n” are passed to the function “swap”.
- These values are copied to formal parameters “a” and “b” in swap function and used.

```
#include<stdio.h>
void swap(int a, int b);    // function prototype, also called function declaration
 int main()
{
   int m = 22, n = 44;
   // calling swap function by value
   printf(" values before swap  m = %d \n and n = %d", m, n);
   swap(m, n);
}
 void swap(int a, int b)
{
   int tmp;
   tmp = a;
   a = b;
   b = tmp;
   printf(" \nvalues after swap m = %d\n and n = %d", a, b);
}
```

Output:

```
values before swap m = 22
and n = 44
values after swap m = 44
and n = 22
```

Call By Reference:

- In call by reference method, the address of the variable is passed to the function as parameter.
- The value of the actual parameter can be modified by formal parameter.
- Same memory is used for both actual and formal parameters since only address is used by both parameters.

Example Program For C Function (Using Call By Reference):

- In this program, the address of the variables “m” and “n” are passed to the function “swap”.
- These values are not copied to formal parameters “a” and “b” in swap function.
- Because, they are just holding the address of those variables.
- This address is used to access and change the values of the variables.

```
#include<stdio.h>
void swap(int *a, int *b);  // function prototype, also called function declaration
int main()
{
   int m = 22, n = 44;
   //  calling swap function by reference
   printf("values before swap m = %d \n and n = %d",m,n);
   swap(&m, &n);
}
 void swap(int *a, int *b)
{
   int tmp;
   tmp = *a;
   *a = *b;
   *b = tmp;
   printf("\n values after swap a = %d \nand b = %d", *a, *b);
}
```

Output

```
values before swap m = 22 and n = 44
values after swap a = 44 and b = 22
```

Function returning non-integers

Many numerical functions like sqrt, sin, and cos return double; other specialized functions return other types than int. To illustrate how to deal with this, let us write and use the function atof(s), which converts the strings to its double-precision floating-point equivalent. It handles an optional sign and decimal point, and the presence or absence of either part or fractional part. Our version is not a high-quality input conversion routine; that would take more space than we care to use. The standard library includes an atof; the header <stdlib.h> declares it.

First, atof itself must declare the type of value it returns, since it is not int. The type name precedes the function name:

```
#include <ctype.h>

  /* atof:  convert string s to double */
  double atof(char s[])
  {
     double val, power;
     int i, sign;

     for (i = 0; isspace(s[i]); i++)  /* skip white space */
         ;
     sign = (s[i] == '-') ? -1: 1;
     if (s[i] == '+' || s[i] == '-')
         i++;
     for (val = 0.0; isdigit(s[i]); i++)
         val = 10.0 * val + (s[i] - '0');
     if (s[i] == '.')
         i++;
     for (power = 1.0; isdigit(s[i]); i++) {
         val = 10.0 * val + (s[i] - '0');
         power *= 10;
     }
     return sign * val / power;
  }
```

Second, and just as important, the calling routine must know that atof returns a non-int value. One way to ensure this is to declare atof explicitly in the calling routine. The declaration is shown in this primitive calculator (barely adequate for check-book balancing), which reads one number per line, optionally preceded with a sign, and adds them up, printing the running sum after each input:

```
#include <stdio.h>

  #define MAXLINE 100

  /* rudimentary calculator */
  main()
  {
     double sum, atof(char []);
```

```
      char line[MAXLINE];
      int getline(char line[], int max);
      sum = 0;
      while (getline(line, MAXLINE) > 0)
         printf("\t%g\n", sum += atof(line));
      return 0;
   }
```

The declaration

```
double sum, atof(char []);
```

says that sum is a double variable, and that atof is a function that takes one char[] argument and returns a double.

The function atof must be declared and defined consistently. If atof itself and the call to it in main have inconsistent types in the same source file, the error will be detected by the compiler. But if (as is more likely) atof were compiled separately, the mismatch would not be detected, atof would return a double that main would treat as an int, and meaningless answers would result.

In the light of what we have said about how declarations must match definitions, this might seem surprising. The reason a mismatch can happen is that if there is no function prototype, a function is implicitly declared by its first appearance in an expression, such as

```
  sum += atof(line)
```

If a name that has not been previously declared occurs in an expression and is followed by left parentheses, it is declared by context to be a function name, the function is assumed to return an int, and nothing is assumed about its arguments. Furthermore, if a function declaration does not include arguments, as in

```
  double atof();
```

that too is taken to mean that nothing is to be assumed about the arguments of atof; all parameter checking is turned off. This special meaning of the empty argument list is intended to permit older C programs to compile with new compilers. But it's a bad idea to use it with new C programs. If the function takes arguments, declare them; if it takes no arguments, use void.

Given atof, properly declared, we could write atoi (convert a string to int) in terms of it:

```
  /* atoi:  convert string s to integer using atof */
  int atoi(char s[])
  {
      double atof(char s[]);
```

```
    return (int) atof(s);
  }
```

Notice the structure of the declarations and the return statement. The value of the expression in

```
  return expression;
```

is converted to the type of the function before the return is taken. Therefore, the value of atof, a double, is converted automatically to int when it appears in this return, since the function atoi returns an int. This operation does potentionally discard information, however, so some compilers warn of it. The cast states explicitly that the operation is intended, and suppresses any warning.

Argument, return value

All functions can be called either with arguments or without arguments in a C program. These functions may or may not return values to the calling function. Now, we will see simple example C programs for each one of the below.

1. C function with arguments (parameters) and with return value.
2. C function with arguments (parameters) and without return value.
3. C function without arguments (parameters) and without return value.
4. C function without arguments (parameters) and with return value.

C functions aspects	syntax

1. With arguments and with return values	**function declaration:**
	int function (int);**function call:** function (a);
	function definition:
	int function(int a)
	{
	statements;
	return a;
	}

2. With arguments and without return values	**function declaration:**
	void function (int);**function call:** function(a);
	function definition:
	void function(int a)
	{
	statements;
	}

3. Without arguments and without return values	**function declaration:**
	void function();**function call:** function();
	function definition:
	void function()
	{
	statements;
	}

4. Without arguments and with return values	**function declaration:**
	int function ( );**function call:** function ( );
	function definition:
	int function( )
	{
	statements;
	return a;
	}

NOTE:

- ➢ If the return data type of a function is "void", then, it can't return any values to the calling function.
- ➢ If the return data type of the function is other than void such as "int, float, double etc", then, it can return values to the calling function.

1. **Example Program For With Arguments & With Return Value:**

In this program, integer, array and string are passed as arguments to the function. The return type of this function is "int" and value of the variable "a" is returned from the function. The values for array and string are modified inside the function itself.

```
#include<stdio.h>
#include<string.h>
int function(int, int[], char[]);
 int main()
{
      int i, a = 20;
      int arr[5] = {10,20,30,40,50};
      char str[30] = "\"fresh2refresh\"";
       printf("    ***values before modification***\n");
      printf("value of a is %d\n",a);
       for (i=0;i<5;i++)
      {
        // Accessing each variable
        printf("value of arr[%d] is %d\n",i,arr[i]);
      }
      printf("value of str is %s\n",str);
      printf("\n    ***values after modification***\n");
      a= function(a, &arr[0], &str[0]);
      printf("value of a is %d\n",a);
      for (i=0;i<5;i++)
      {
        // Accessing each variable
        printf("value of arr[%d] is %d\n",i,arr[i]);
      }
      printf("value of str is %s\n",str);
      return 0;
}
int function(int a, int *arr, char *str)
{
    int i;
     a = a+20;
    arr[0] = arr[0]+50;
    arr[1] = arr[1]+50;
    arr[2] = arr[2]+50;
    arr[3] = arr[3]+50;
    arr[4] = arr[4]+50;
    strcpy(str,"\"modified string\"");
     return a;
}
```

Output:

```
***values before modification***
value of a is 20
value of arr[0] is 10
value of arr[1] is 20
value of arr[2] is 30
value of arr[3] is 40
value of arr[4] is 50
value of str is “fresh2refresh”***values after modification***
value of a is 40
value of arr[0] is 60
value of arr[1] is 70
value of arr[2] is 80
value of arr[3] is 90
value of arr[4] is 100
value of str is “modified string”
```

2. Example Program For With Arguments & Without Return Value:

In this program, integer, array and string are passed as arguments to the function. The return type of this function is "void" and no values can be returned from the function. All the values of integer, array and string are manipulated and displayed inside the function itself.

```
#include<stdio.h>
void function(int, int[], char[]);
 int main()
{
     int a = 20;
     int arr[5] = {10,20,30,40,50};
     char str[30] = "\"vemu\"";
      function(a, &arr[0], &str[0]);
     return 0;
}
void function(int a, int *arr, char *str)
{
   int i;
   printf("value of a is %d\n\n",a);
   for (i=0;i<5;i++)
     {
       // Accessing each variable
       printf("value of arr[%d] is %d\n",i,arr[i]);
     }
   printf("\nvalue of str is %s\n",str);
}
```

Output:

```
value of a is 20
 value of arr[0] is 10
value of arr[1] is 20
value of arr[2] is 30
value of arr[3] is 40
value of arr[4] is 50
 value of str is “vemu”
```

3. Example Program For Without Arguments & Without Return Value:

In this program, no values are passed to the function "test" and no values are returned from this function to main function.

```
#include<stdio.h>
void test();
```

```
 int main()
{
   test();
   return 0;
}
 void test()
{
   int a = 50, b = 80;
   printf("\nvalues : a = %d and b = %d", a, b);
}
```

Output:

```
values : a = 50 and b = 80
```

4. Example Program For Without Arguments & With Return Value:

In this program, no arguments are passed to the function “sum”. But, values are returned from this function to main function. Values of the variable a and b are summed up in the function “sum” and the sum of these value is returned to the main function.

```
#include<stdio.h>
int sum();
int main()
{
   int addition;
   addition = sum();
   printf("\nSum of two given values = %d", addition);
   return 0;
}
int sum()
{
   int a = 50, b = 80, sum;
   sum = a + b;
   return sum;
}
```

Output:

```
Sum of two given values = 130
```

Do You Know How Many Values Can Be Return From C Functions?

- Always, Only one value can be returned from a function.
- If you try to return more than one values from a function, only one value will be returned that appears at the right most place of the return statement.

- For example, if you use "return a,b,c" in your function, value for c only will be returned and values a, b won't be returned to the program.
- In case, if you want to return more than one values, pointers can be used to directly change the values in address instead of returning those values to the function.

Block Structure

Blocks are fundamental to structured programming, where control **structures** are formed from **blocks**. if you had ever programmed in assembler language, it would be obvious. Because statements in **C** are grouped in **blocks**, delimited by curly braces.**C** programs are composed of **blocks** of statements

Preprocessor Directives

The C preprocessor is a macro processor that is used automatically by the C compiler to transform your program before actual compilation (Proprocessor direcives are executed before compilation.). It is called a macro processor because it allows you to define macros, which are brief abbreviations for longer constructs. A macro is a segment of code which is replaced by the value of macro. Macro is defined by #define directive.
Preprocessing directives are lines in your program that start with #. The # is followed by an identifier that is the directive name. For example, #define is the directive that defines a macro. Whitespace is also allowed before and after the #.
The # and the directive name cannot come from a macro expansion. For example, if foo is defined as a macro expanding to define, that does not make #foo a valid preprocessing directive.
All preprocessor directives starts with hash # symbol
List of preprocessor directives

1. #include
2. #define
3. #undef
4. #ifdef
5. #ifndef
6. #if
7. #else
8. #elif
9. #endif
10. #error
11. #pragma

1. #include

The #include preprocessor directive is used to paste code of given file into current file. It is used include system-defined and user-defined header files. If included file is not found, compiler renders error. It has three variants
#include <file>
This variant is used for system header files. It searches for a file named file in a list of directories specified by you, then in a standard list of system directories.

#include "file"

This variant is used for header files of your own program. It searches for a file named file first in the current directory, then in the same directories used for system header files. The current directory is the directory of the current input file.

#include anything else

This variant is called a computed #include. Any #include directive whose argument does not fit the above two forms is a computed include.

2. Macro's (#define)

Let's start with macro, as we discuss, a macro is a segment of code which is replaced by the value of macro. Macro is defined by #define directive.

Syntax: #define token value

There are two types of macros:

1. Object-like Macros
2. Function-like Macros

Object-like Macros

The object-like macro is an identifier that is replaced by value. It is widely used to represent numeric constants

Syntax: #define PI 3.1415

Here, PI is the macro name which will be replaced by the value 3.14. Let's see an example of Object-like Macros

```
#include <stdio.h>
#define PI 3.1415
main()
{
printf("%f",PI);
}
```

Output: 3.140003.1400

Function-like Macros

The function-like macro looks like function call.

```
#define MIN(a,b) ((a)<(b)?(a):(b))
```

Here, MIN is the macro name. Let's see an example of Function

```
#include <stdio.h>
#define MIN(a,b) ((a)<(b)?(a):(b))
void main()
{
printf("Minimum between 10 and 20 is: %d\n", MIN(10,20));
}
```

Output: minimum between 10 and 20 is: 10

Preprocessor Formatting

A preprocessing directive cannot be more than one line in normal circumstances. It may be split cosmetically with Backslash-Newline. Comments containing Newlines can also divide the directive into multiple lines.

for example, you can split a line cosmetically with Backslash-Newline anywhere

```
/*
*/#/*
*/defi\
Ne FO\
0 10\
20
```

is equivalent into #define FOO 1020

3. #undef

To undefine a macro means to cancel its definition. This is done with the #undefdirective.

```
#undef token
```

define and undefine example

```
#include <stdio.h>
#define PI 3.1415
#undef PI
main() {
printf("%f",PI);
}
```

Output: Compile Time Error: 'PI' undeclared

4. #ifdef

The #ifdef preprocessor directive checks if macro is defined by #define. If yes, it executes the code.

Syntax:

```
        #ifdef MACRO
        //code
        #endif
```

5. #ifndef

The #ifndef preprocessor directive checks if macro is not defined by #define. If yes, it executes the code.

Syntax:

```
        #ifdef MACRO
        //code
        #endif
```

6. #if

The #if preprocessor directive evaluates the expression or condition. If condition is true, it executes the code.

Syntax:

```
#if expression
//code
#endif
```

7. #else

The #else preprocessor directive evaluates the expression or condition if condition of #if is false. It can be used with #if, #elif, #ifdef and #ifndef directives.

```
#if expression
//if code
#else
//else code
#endif
```

8. #error

The #error preprocessor directive indicates error. The compiler gives fatal error if #error directive is found and skips further compilation process.

9. #pragma

The #pragma preprocessor directive is used to provide additional information to the compiler. The #pragma directive is used by the compiler to offer machine or operating-system feature. Different compilers can provide different usage of #pragma directive.

Recursion

Recursion is the process which comes into existence when a function calls a copy of itself to work on a smaller problem. Any function which calls itself is called recursive function, and such function calls are called recursive calls. Recursion involves several numbers of recursive calls. However, it is important to impose a termination condition of recursion. Recursion code is shorter than iterative code however it is difficult to understand.

Recursion cannot be applied to the entire problem, but it is more useful for the tasks that can be defined in terms of similar subtasks. For Example, recursion may be applied to sorting, searching, and traversal problems.

Generally, iterative solutions are more efficient than recursion since function call is always overhead. Any problem that can be solved recursively, can also be solved iteratively. However, some problems are best suited to be solved by the recursion, for example, tower of Hanoi, Fibonacci series, factorial finding, etc.

```
#include <stdio.h>
int fact (int);
int main()
{
   int n,f;
   printf("Enter the number whose factorial you want to calculate?");
   scanf("%d",&n);
   f = fact(n);
   printf("factorial = %d",f);
}
int fact(int n)
{
   if (n==0)
   {
       return 0;
   }
   else if ( n == 1)
   {
       return 1;
   }
   else
   {
       return n*fact(n-1);
   }
}
```

Output

```
Enter the number whose factorial you want to calculate?5
factorial = 120
```

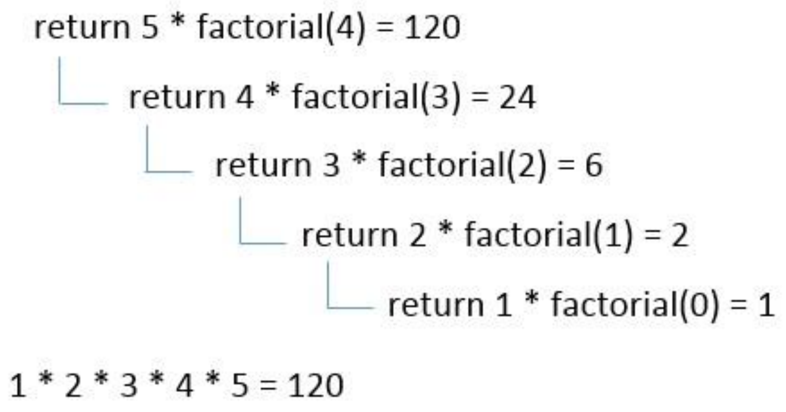

return 5 * factorial(4) = 120
return 4 * factorial(3) = 24
return 3 * factorial(2) = 6
return 2 * factorial(1) = 2
return 1 * factorial(0) = 1
1 * 2 * 3 * 4 * 5 = 120

Fig: Recursion

Recursive Function

A recursive function performs the tasks by dividing it into the subtasks. There is a termination condition defined in the function which is satisfied by some specific subtask. After this, the recursion stops and the final result is returned from the function.

The case at which the function doesn't recur is called the base case whereas the instances where the function keeps calling itself to perform a subtask, is called the recursive case. All the recursive functions can be written using this format.

Pseudocode for writing any recursive function is given below.

```
if (test_for_base)
{
    return some_value;
}
else if (test_for_another_base)
{
   return some_another_value;
}
else
{
   // Statements;
   recursive call;
}
```

Example of recursion

```
#include<stdio.h>
int fibonacci(int);
void main ()
{
   int n,f;
   printf("Enter the value of n?");
   scanf("%d",&n);
   f = fibonacci(n);
   printf("%d",f);
}
int fibonacci (int n)
{
   if (n==0)
   {
  return 0;
   }
   else if (n == 1)
   {
      return 1;
   }
   else
   {
      return fibonacci(n-1)+fibonacci(n-2);
   }
}
```

Output

```
Enter the value of n? 12 144
```

Memory allocation of Recursive method

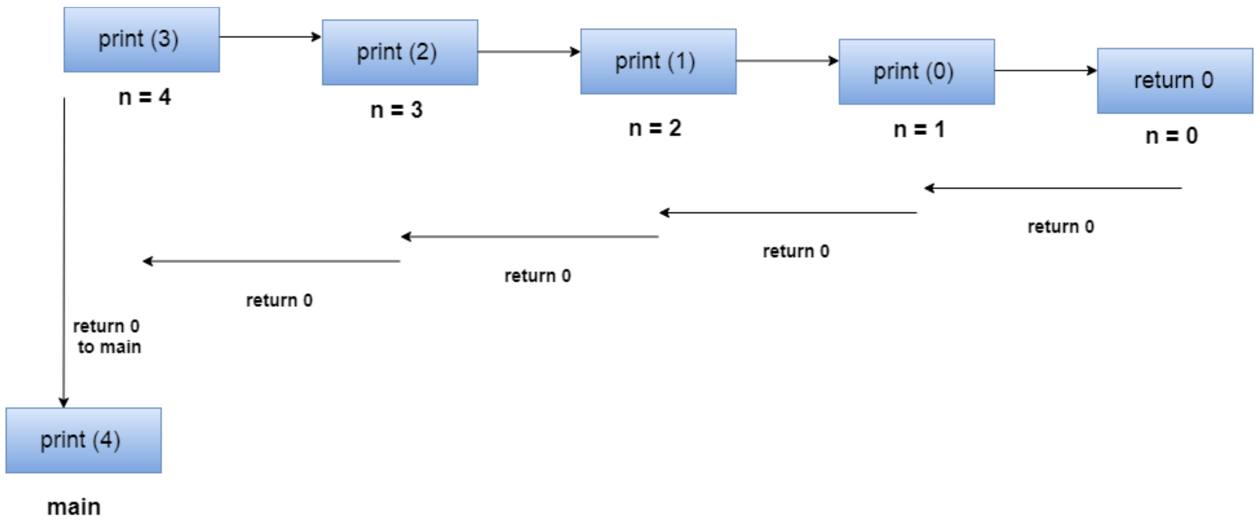
Each recursive call creates a new copy of that method in the memory. Once some data is returned by the method, the copy is removed from the memory. Since all the variables and other stuff declared inside function get stored in the stack, therefore a separate stack is maintained at each recursive call.

Once the value is returned from the corresponding function, the stack gets destroyed. Recursion involves so much complexity in resolving and tracking the values at each recursive call. Therefore we need to maintain the stack and track the values of the variables defined in the stack.

Let us consider the following example to understand the memory allocation of the recursive functions.

```
int display (int n)
{
   if(n == 0)
     return 0; // terminating condition
   else
   {
     printf("%d",n);
      return display(n-1); // recursive call
   }
}
```

Let us examine this recursive function for n = 4. First, all the stacks are maintained which prints the corresponding value of n until n becomes 0, Once the termination condition is reached, the stacks get destroyed one by one by returning 0 to its calling stack. Consider the following image for more information regarding the stack trace for the recursive functions.



Stack tracing for recursive function call

UNIT-IV

UNIT 4

Factoring methods: Finding the square root of a number, the smallest divisor of a number, the greatest common divisor of two integers, generating prime numbers.

Pointers and arrays: Pointers and addresses, pointers and function arguments, pointers and arrays, address arithmetic, character pointers and functions, pointer array; pointers to pointers, Multi-dimensional arrays, initialization of arrays, pointer vs. multi-dimensional arrays, command line arguments, pointers to functions, complicated declarations.

Array Techniques: Array order reversal, finding the maximum number in a set, removal of duplicates from an order array, finding the kth smallest element.

Write a c program to find the square root of a number.

```
#include <stdio.h>
#include <math.h>
int main()
{
   double num = 6, squareRoot;
   squareRoot =  sqrt(num);
   printf("Square root of %lf =  %lf", num, squareRoot);
   return 0;
}
```

Write c program using recursion for finding the GCD of two numbers.

Source Code:-

```
#include<stdio.h>
#include<conio.h>
void main()
{
        int a,b,r;
        clrscr();
        printf("\n Enter two numbers:");
        scanf("%d%d",&a,&b);
        if(a>b)
          r=rgcd(a,b);
        else
          r=rgcd(b,a);
        printf("GCD of %3d and %3d is=%3d",a,b,r);
        getch();
}
int rgcd(int x,int y)
{
        if (x%y==0)
          return y;
        else
          return rgcd(y,(x%y));
}
```

Input:-

```
Enter two numbers:8 2
```

Output:-

```
 GCD of   8 and   2 is=  2
```

Write a program to check whether the number is prime or not.

Sour

```
#include<stdio.h>
#include<conio.h>
void main()
{
        int n,i,k=0;
        clrscr();
        printf("\n Enter a number :");
        scanf("%d",&n);
        i=1;
        while(i<=n)
        {
                if(n%i==0)
                  k++;
                i++;
        }
        if(k==2)
                printf("\n Given number is a prime number");
        else
                printf("\n Given number is not a prime number");
}
```

Input:

```
 Enter a number: 6
```

Output:

```
 Given number is not a prime number
```

Input:

```
Enter a number: 11
```

Output:

```
  Given number is a prime number
```

Write a program to find the series of prime numbers in the given range

Source Code:-

```
#include<stdio.h>
#include<conio.h>
void  main()
{
          int n,i,j,k=0;
          clrscr();
          printf("Enter n value upto where prime numbers to be found: ");
          scanf("%d",&n);
          if(n==0)
          {
                    printf(" \n Invalid number entered");
          }
          printf(“ \n Required prime numbers are :”);
          i=2;
          while(i<=n)
          {
                    k=0;
                    j=2;
                    while(j<n)
                    {

                    if(i%j==0)
                    k+=1;
                      j++;
                    }
                    if(k==1)
                          printf("%4d" , i);
                    i++;
          }
}
```

Input:-
Enter n value upto where prime numbers to be found: 10

Output:
Required prime numbers are:
 2 3 5 7

Pointers and address:

Define a Pointer

A pointer is a variable that holds a memory address. This address is the location of another object (typically another variable) in memory.

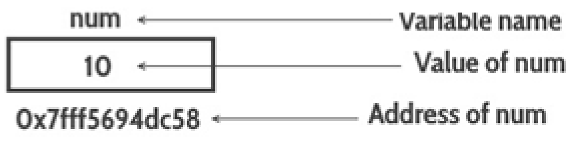
For example: - if one variable contains the address of another variable, the first variable is said to point to the second. Figure illustrates this situation.

Pointer Variable Declaration:-

A pointer declaration consists of a base type, an *, and the variable name. The general form for declaring a pointer variable is.

Syntax:- **Data type *variable_name;**

Example: int num=10;



- Type is the base type of the pointer and may be any valid type.
- The name of the pointer variable is specified by name.
- The base type of the pointer defines the type of object to which the pointer will point.
- Technically, any type of pointer can point anywhere in memory.
- All pointer operations are done relative to the pointer's base type.

For example:- when you declare a pointer to be of type int *, the compiler assumes that any address that it holds points to an integer whether it actually does or not. (That is, an int * pointer always "thinks" that it points to an int object, no matter what that piece of memory actually contains.)

Therefore, when you declare a pointer, you must make sure that its type is compatible with the type of object to which you want to point.

Pointer Operators

There are two pointer operators:

- **“Address of”(&) Operator**
- **“Value at Address”(*) Operator**

- The & is a unary operator that returns the memory address of its operand. (A unary operator only requires one operand.)

```
For example: -      m = &count;
                    &= “address of”
```

- Places into m the memory address of the variable count.
- This address is the computer's internal location of the variable. It has nothing to do with the value of count.
- Therefore, the preceding assignment statement can be verbalized as "m receives the address of count.”
- Assume that the variable count uses memory location 2000 to store its value and count has a value of 100.
- Then, after the preceding assignment, m will have the value 2000.

The second pointer operator: - *, is the complement of &.

- It is a unary operator that returns the value located at the address that follows. For example, if m contains the memory address of the variable count,

```
q = *m;            * -- "at address."
```

- Places the value of count into q. Thus, q will have the value 100 because 100 is stored at location 2000, which is the memory address that was stored in m.
- In this case, the preceding statement can be verbalized as "q receives the value at address m."

Pointer Initialization: is the process of assigning address of a variable to pointer variable.

- Pointer variable contains address of variable of same data type.
- In C language address operator & is used to determine the address of a variable.
- The & (immediately preceding a variable name) returns the address of the variable associated with it.

```
int a = 10 ;
int *ptr ;          //pointer declaration
ptr = &a ;          //pointer initialization or,
int *ptr = &a ;     //initialization and declaration together Pointer variable always points
                    to same type of data.
```

Example will clearly explain the initialization of Pointer Variable.

```
#include<stdio.h>
int main()
        {
        int a;// Step 1
        int *ptr;  // Step 2
        a = 10;       // Step 3
         ptr = &a; // Step 4
        return(0);
        }
```

Explanation of Above Program:

- Pointer should not be used before initialization.
- "ptr" is pointer variable used to store the address of the variable.
- Stores address of the variable 'a'.
- Now "ptr" will contain the address of the variable "a".

Note: Pointers are always initialized before using it in the program

A Simple Example of Pointers

```
#include <stdio.h>
int main()
{
   int num = 10;
   printf("Value of variable num is: %d", num);
   printf("\nAddress of variable num is: %p", &num);
   return 0;
}
```

Output:

```
Value of variable num is: 10
Address of variable num is: 0x7fff5694dc58
```

A Simple Example of Pointers

```
#include <stdio.h>
int main()
{
   //Variable declaration
   int num = 10;

   //Pointer declaration
   int *p;

   //Assigning address of num to the pointer p
   p = &num

   printf("Address of variable num is: %p", p);
   return 0;
}
```

Output:

```
Address of variable num is: 0x7fff5694dc58
```

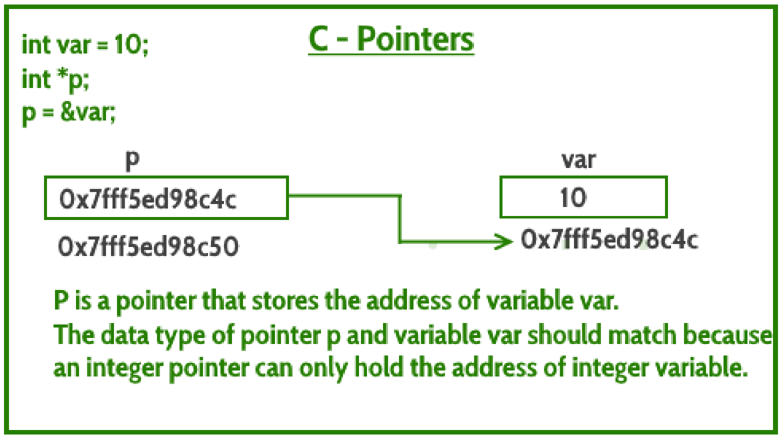
Example of Pointer demonstrating the use of & and *

```
#include <stdio.h>
int main()
{
   /* Pointer of integer type, this can hold the
    * address of a integer type variable.  */
   int *p;
```

```
  int var = 10;

                        /* Assigning the address of variable var to the pointer
                         * p. The p can hold the address of var because var is
                         * an integer type variable. */
  p= &var;
  printf("Value of variable var is: %d", var);
  printf("\nValue of variable var is: %d", *p);
  printf("\nAddress of variable var is: %p", &var);
  printf("\nAddress of variable var is: %p", p);
  printf("\nAddress of pointer p is: %p", &p);
  return 0;
}
```

Output:

```
        Value of variable var is: 10
        Value of variable var is: 10
        Address of variable var is: 0x7fff5ed98c4c
        Address of variable var is: 0x7fff5ed98c4c
        Address of pointer p is: 0x7fff5ed98c50
```


C - Pointers
int var = 10;
int *p;
p = &var;
p
0x7fff5ed98c4c
0x7fff5ed98c50
var
10
0x7fff5ed98c4c
P is a pointer that stores the address of variable var.
The data type of pointer p and variable var should match because an integer pointer can only hold the address of integer variable.

POINTERS AND FUNCTION ARGUMENTS:

Pointer as a function parameter is used to hold addresses of arguments passed during function call. This is also known as call by reference. When a function is called by reference any change made to the reference variable will affect the original variable.

Example Time: Swapping two numbers using Pointer

```
#include <stdio.h>
void swap(int *a, int *b);
int main()
{
   int m = 10, n = 20;
   printf("m = %d\n", m);
   printf("n = %d\n\n", n);
   swap(&m, &n);    //passing address of m and n to the swap function
   printf("After Swapping:\n\n");
   printf("m = %d\n", m);
   printf("n = %d", n);
   return 0;
}
```

```
/*   pointer 'a' and 'b' holds and
    points to the address of 'm' and 'n' */

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

Output

```
m = 10
n = 20
After Swapping:
m = 20
n = 10
```

Functions returning Pointer variables

A function can also return a pointer to the calling function. In this case you must be careful, because local variables of function don't live outside the function. They have scope only inside the function. Hence if you return a pointer connected to a local variable, that pointer will be pointing to nothing when the function ends.

```
#include <stdio.h>
int* larger(int*, int*);
void main()
{
    int a = 15;
    int b = 92;
    int *p;
    p = larger(&a, &b);
    printf("%d is larger",*p);
}
int* larger(int *x, int *y)
{
    if(*x > *y)
        return x;
    else
        return y;
}
```

Output

```
92 is larger
```

Pointer to functions: - It is possible to declare a pointer pointing to a function which can then be used as an argument in another function. A pointer to a function is declared as follows,

```
type (*pointer-name)(parameter);
```

Here is an example:

```
int (*sum)();   //legal declaration of pointer to function
int *sum();     //This is not a declaration of pointer to function
```

A function pointer can point to a specific function when it is assigned the name of that function.

```
int sum(int, int);
int (*s)(int, int);
s = sum;
```

Here s is a pointer to a function sum. Now sum can be called using function pointer s along with providing the required argument values.

```
s (10, 20);
```

Example of Pointer to Function

```
#include <stdio.h>
int sum(int x, int y)
{
   return x+y;
}
int main( )
{
   int (*fp)(int, int);
   fp = sum;
   int s = fp(10, 15);
   printf("Sum is %d", s);

   return 0;
}
```

Output

```
25
```

Pointers and Arrays

There is a close relationship between pointers and arrays. Consider this program fragment: char str[80], *p1; p1 = str;

- Here, p1 has been set to the address of the first array element in str.
- To access the fifth element in str, make it as str[4] or *(p1+4)
- Both statements will return the fifth element.
- Remember, arrays start at 0.
- To access the fifth element, you must use 4 to index str.
- You also add 4 to the pointer p1 to access the fifth element because p1 currently points to the first element of **str.**

- The array name without an index returns the starting address of the array, which is the address of the first element.
- C provides two methods of accessing array elements: pointer arithmetic and array indexing.
- C programmers often use pointers to access array elements.
- These two versions of putstr( ) one with array indexing and one with pointers illustrate how you can use pointers in place of array indexing. The putstr( ) function writes a string to the standard output device one character at a time.

```
/* Index s as an array. */
void putstr(char *s)
        {
                register int t;
```

```
                    for(t=0; s[t]; ++t) putchar(s[t]);
                     }
                    /* Access s as a
                    pointer. */
                    void putstr(char *s)
                    {
                    while(*s)putchar(*s++);
                }
```

Pointer Arithmetic/ Address Arithmetic

There are only two arithmetic operations that you can use on pointers:

- **Addition**
- **Subtraction.**

To understand what occurs in pointer arithmetic, let p1 be an integer pointer with a current value of 2000. Also, assume ints are 2 bytes long.
After the expression
p1++;
p1 contains 2002, not 2001.

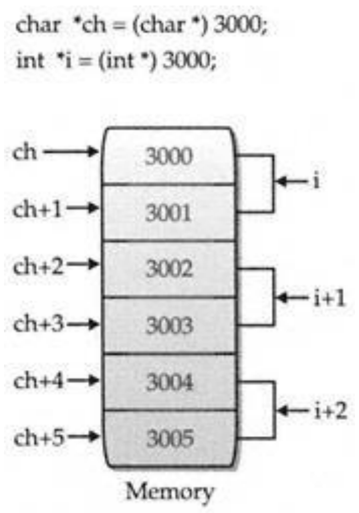
The reason for this is that each time p1 is incremented it will point to the next integer. The same is true of decrements. For example, assuming that p1 has the value 2000, the expression p1--; causes p1 to have the value 1998.
Generalizing from the preceding example, the following rules govern pointer arithmetic.

- Each time a pointer is incremented, it points to the memory location of the next element of its base type.
- Each time it is decremented, it points to the location of the previous element.
- When applied to char pointers, this will appear as "normal" arithmetic because a char object is always 1 byte long no matter what the environment.
- All other pointers will increase or decrease by the length of the data type they point to.
- This approach ensures that a pointer is always pointing to an appropriate element of its base type. Below figure illustrates this concept.

We may add or subtract integers to or from pointers.
The expression p1 = p1 + 12; makes p1 point to the 12th element of p1's type beyond the one it currently points to. Besides addition and subtraction of a pointer and an integer, only one other arithmetic operation is allowed, you can subtract one pointer from another in order to find the number of objects of their base type that separate the two.

All other arithmetic operations are prohibited, such as

- We cannot multiply or divide pointers;
- We cannot add two pointers;
- We cannot apply the bitwise operators to them; and
- We cannot add or subtract type float or double to or from pointers.



All pointer arithmetic is relative to its base type (assume 2-byte integers)

Character Pointers and Functions:

Let’s say we have a char variable ch and a pointer ptr that holds the address of ch.

```
char ch='a';
char *ptr;
```
Read the value of ch
```
printf("Value of ch: %c", ch);
```
or
```
printf("Value of ch: %c", *ptr);
```

Change the value of ch

```
ch = 'b';
```
or
```
*ptr = 'b';
```
The above code would replace the value ‘a’ with ‘b’.

Can you guess the output of following C program?

```
#include <stdio.h>
int main()
{
   int var =10;
   int *p;
   p= &var;

   printf ( "Address of var is: %p", &var);
   printf ( "\nAddress of var is: %p", p);

   printf ( "\nValue of var is: %d", var);
   printf ( "\nValue of var is: %d", *p);
   printf ( "\nValue of var is: %d", *( &var));

   /* Note I have used %p for p's value as it represents an address*/
   printf( "\nValue of pointer p is: %p", p);
   printf ( "\nAddress of pointer p is: %p", &p);
```

```
    return 0;
}
```

Output:

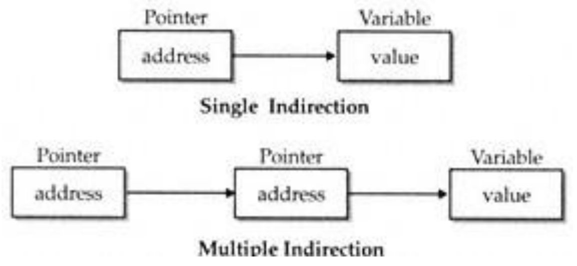
```
Address of var is: 0x7fff5d027c58
Address of var is: 0x7fff5d027c58
Value of var is: 10
Value of var is: 10
Value of var is: 10
Value of pointer p is: 0x7fff5d027c58
Address of pointer p is: 0x7fff5d027c50
```

Pointers to Pointers/ Multiple Indirection

- When a pointer point to another pointer that points to the target value , then that pointer is called multiple indirection, or pointers to pointers.
- Below figure shows the concept of multiple indirection.
- The value of a normal pointer is the address of the object that contains the desired value.
- In the case of a pointer to a pointer, the first pointer contains the address of the second pointer, which points to the object that contains the desired value.
- Multiple indirections can be carried on to whatever extent desired, but more than a pointer to a pointer is rarely needed.
- In fact, excessive indirection is difficult to follow and prone to conceptual errors.
- A variable that is a pointer to a pointer must be declared as such. This can be done by placing an additional asterisk in front of the variable name.

For example, the following declaration tells the compiler that new balance is a pointer to a pointer of type float: float **newbalance;

the new balance is not a pointer to a floating-point number but rather a pointer to a float pointer.



To access the target value indirectly pointed to by a pointer to a pointer, you must apply the asterisk operator twice, as in this example:

```
#include
<stdio.h>
int
main(void)
        {
                int x, p, **q;
                x = 10;
                p = &x;
                q = &p;
                printf("%d", **q); /* print the value of x */
                return 0;
```

}

Here, p is declared as a pointer to an integer and q as a pointer to a pointer to an integer. The call to **printf( )** prints the number 10 on the screen.

Array: An array is a collection of two or more adjacent cells or elements of similar type.

- Each cell in an array is called as array element.
- Each array should be identified with a meaningful name and the number of adjacent cells to be associated for that array should be specified.

Single/One-Dimensional Arrays

The one dimensional arrays are also known as Single dimension array and is a type of Linear Array. In the one dimension array the data type is followed by the array name which is further followed by the size (an integer value) enclosed in square brackets and this represents the dimension either row wise or column wise.

Declaration of Array: To declare an array the following syntax should be followed:

```
<data type>  arrayname[SIZE];
```

- Where data type can be any fundamental or basic or intrinsic or implicit or user defined data type.
- Array name should follow the rules of an identifier.
- SIZE takes an integer value that specifies the number of adjacent cells required for that array.

For e.g., int number[10];

Where number is the name of the array, the integer value 10 indicates the size of the array i.e., 10 adjacent cells are allocated with each cell storing integer element as the data type is int.

The memory is allocated with index starting from zero and ends with its size-1. Pictorially the memory allocation and its identification is represented as shown below:

Array name	Number[0]	Number[1]	..	..	..	..	..	..	Number[9]
Element									

Initializing elements to the array:

Elements can be initialized to the array as shown below. We initialize values to the array only if the values are going to be fixed.

General syntax:

```
<data type> arrayname[SIZE]={value1,value2,….,valueN};
```

For e.g. for the above declarations we will see how to initialize (assigning) values. int number[10]={10,20,30,40,50,60,70,80,90,12};

(Or)

number[0]=10;
number[1]=20;
number[2]=30;
…
.

number [9]=12;

Where 10 is assigned to location number [0], 20 is assigned to location number [1] and so on.

Referencing the Array:

An array can be referenced in the program in any of the executable statements in either one of the ways given below:

Way 1:

Syntax is given below: arrayname[index];

Where index takes an integer value, this integer value specifies to the array location which is index + 1, this integer value also should not be greater than the size of that array specified in the declaration of the array.

For e.g., int number [15];

to refer 11th location of this array we go for number[10]

Way 2: Syntax is given blow:

variable=value;

arrayname[variable];

Where an integer value is to be initialized to a variable and that variable is used within the square brackets of that array and this variable is called as subscript variable as this variable contains an integer value that refers to the index of that array.

Way 3: Using for loops for sequential access:

In order to read elements into the array sequentially or to access elements of an array sequentially we make use of looping constructs. We will see how this is done.

For reading elements into the array sequentially:

For e.g. for the declaration given below, we will see how elements are read sequentially into the array. int number[10]; /* array declaration */

```
/* reading elements into the array sequentially */
for ( i=0; i < 10 ; i++ )
scanf( " %d ", & number[ i ] );
```

Explanation:

For loop is used to vary the index of the array from 0 to its size-1.The variable used in for loop is used in the scanf statement along with the array name enclosed in square brackets. This variable is called as subscript variable.

For printing elements of the array sequentially:

For e.g. if we suppose assume that the array is declared and the elements are already initialized, then we will see how elements are accessed sequentially from the array.

```
/* accessing elements from the array
sequentially */ for( i=0; i<10 ;i++ )
{
        Printf(" %d ", number[i]);
}
```

Explanation:

For loop is used to vary the index of the array from 0 to its size-1.The variable used in the for loop is used in any of the executable statement along with the array name enclosed in square brackets.In the above example each and every element of the array is accessed and assigned to the variable num and that value is immediately printed.

/* Sample program to illustrate the declaration, initialization and accessing elements of the array */

```
#include <stdio.h>
void main()
{
        int number[10]={1,2,3,4,5,6,7,8,9,10}; /* array declaration and intialization*/
         int i; clrscr();
        /* printing the elements of the
        array */ printf("elements in the
        array are:\n"); for(i=0; i<10; i++)
        printf("%d\n",number[i]);
}
```

/* Sample program to illustrate the declaration, reading and printing elements of the array */

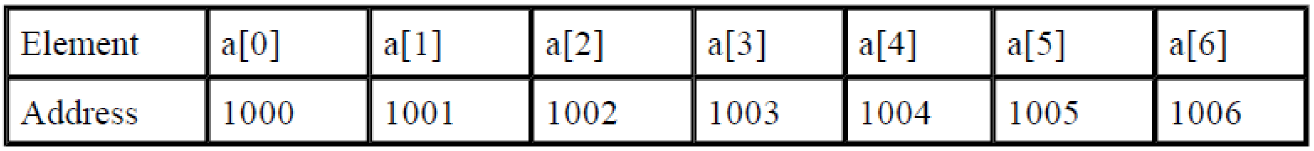
```
#include <stdio.h>
void main()
{
        int number[10] /* array declaration */
        int i; clrscr();
        /* reading the elements of the array */
        printf("Enter the elements into the array are:\n");
        for(i=0; i<10; i++)
        scanf(" %d ", &number[i]);
        /* printing the elements of the array */
        printf("elements in the array are:\n");
        for(i=0; i<10; i++)
        printf("%d\n",number[i]);
}
```

Generating a Pointer to an Array

You can generate a pointer to the first element of an array by simply specifying the array name, without any index.

For example, given

```
int sample[10];
```



Element	a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]
Address	1000	1001	1002	1003	1004	1005	1006

A seven-element character array beginning at location 1000

You can generate a pointer to the first element by using the name sample. Thus, the following program fragment assigns p the address of the first element of sample:

```
int *p;
int sample[10];
 p = sample;
```

You can also specify the address of the first element of an array by using the & operator. For example, sample and &sample [0] both produce the same results.

However, in professionally written C code, you will almost never see &sample[0].

Passing Single Dimensional Arrays to Functions:

Arrays can be passed as argument in functions. To pass arrays as arguments one should know how to pass the arguments in function call and how to receive the arguments in the function definition.

```
Function name (array name);
```

The general format to pass an array as argument in function call is given below:
Where array name is the name of the array which is declared in the calling function. The general format to receive the arguments in the function definition is given below.

```
<data type> functionname(<data type> arrayname[ ])
```

Where array name is the name that will be referred in the function definition. This is used in place of the array name passed from the function call and this array should be declared for its type.
For e.g.,
int data[10]; we assume that the elements to the array are either initialized or read from the keyboard , then passing the array as argument in the function call is done as shown below:

printelements(data);

the arguments passed from function call are received in the function definition as shown below:

void printelements(int num[ ])

```
/* sample program illustrating how arrays are passed as arguments in functions */
#include <stdio.h>
void printelements(int [ ]); /* prototype declaration */
void main()
{
        int data[10]; /* array
          declaration */ int i;
          clrscr();
          /* reading elements into the array from
          keyboard */ printf("enter the elements:");
          for(i=0; i<10; i++)
          scanf(" %d ",
          &data[i]);
          /* function call */
          printelements(data); /* array passed as argument in function call */
}
          /* function definition */
void printelements(int num[ ])        /* actual arguments received in function definition */
{
          int i;
```

```
            printf("elements
            are:\n"); for(i=0;
            i<10; i++)
            printf("%d\n",num[
            i]);
    }
```

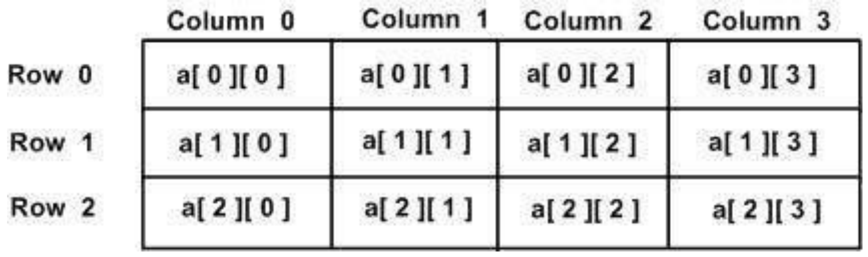
Two-Dimensional Arrays:

The simplest form of the multidimensional array is the two-dimensional array.
A two-dimensional array is, in essence, a list of one-dimensional arrays. To declare a two-dimensional integer array of size x,y you would write something as follows:

```
type arrayName [size1][ size2 ];
```

Where type can be any valid C data type and arrayName will be a valid C identifier.

A two-dimensional array can be think as a table which will have x number of rows and y number of columns. A 2-dimensional array a, which contains three rows and four columns can be shown as below:



	Column 0	Column 1	Column 2	Column 3
Row 0	a[0][0]	a[0][1]	a[0][2]	a[0][3]
Row 1	a[1][0]	a[1][1]	a[1][2]	a[1][3]
Row 2	a[2][0]	a[2][1]	a[2][2]	a[2][3]

Thus, every element in array a is identified by an element name of the form a[i][j], where a is the name of the array, and i and j are the subscripts that uniquely identify each element in a.

Initializing Two-Dimensional Arrays:

Multidimensional arrays may be initialized by specifying bracketed values for each row. Following is an array with 3 rows and each row has 4 columns.

```
int a[3][4] = {   {0, 1, 2, 3} ,          /*  initializers for row indexed by 0  */
                  {4, 5, 6, 7} ,          /*  initializers for row indexed by 1  */
                  {8, 9, 10, 11}          /* initializers for row indexed by 2 */

};
```

The nested braces, which indicate the intended row, are optional. The following initialization is equivalent to previous example:

```
int a[3][4] = {0,1,2,3,4,5,6,7,8,9,10,11};
```

Accessing Two-Dimensional Array Elements:

An element in 2-dimensional array is accessed by using the subscripts, i.e., row index and column index of the array. For example: int val = a[2][3];

The above statement will take 4th element from the 3rd row of the array.

You can verify it in the above diagram. Let us check below program where we have used nested loop to handle a two dimensional array:

```
#include <stdio.h>

int main ()

{

  /* an array with 5 rows and 2 columns*/
  int a[5][2] = { {0,0}, {1,2}, {2,4}, {3,6},{4,8}};

  int i, j;

  /* output each array element's

  value */ for ( i = 0; i < 5; i++ )

  {
    for ( j = 0; j < 2; j++ )

    {

      printf("a[%d][%d] = %d\n", i,j, a[i][j] );
    }

  }

  return 0;
}
```

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

```
a[0][0]: 0
a[0][1]: 0

a[1][0]: 1

a[1][1]: 2
a[2][0]: 2
a[2][1]: 4

a[3][0]: 3
a[3][1]: 6
a[4][0]: 4

a[4][1]: 8
```

As explained above, you can have arrays with any number of dimensions, although it is likely that most of the arrays you create will be of one or two dimensions.

Indexing Pointers

Pointers and arrays are closely related. As you know, an array name without an index is a pointer to the first element in the array.
For example, consider the following array:

```
char p[10];
```

The following statements are identical:

p &p[0]

Put another way,

```
p == &p[0]
```

Evaluates to true because the address of the first element of an array is the same as the address of the array. As stated, an array name without an index generates a pointer.

Conversely, a pointer can be indexed as if it were declared to be an array. For example, consider this program fragment:

```
int *p, i[10];
p = i;
p[5] = 100; /* assign using index */
*(p+5) = 100; /* assign using pointer arithmetic */
```

Both assignment statements place the value 100 in the sixth element of i.

The first statement indexes p; the second uses pointer arithmetic. Either way, the result is the same

Multi-dimensional arrays

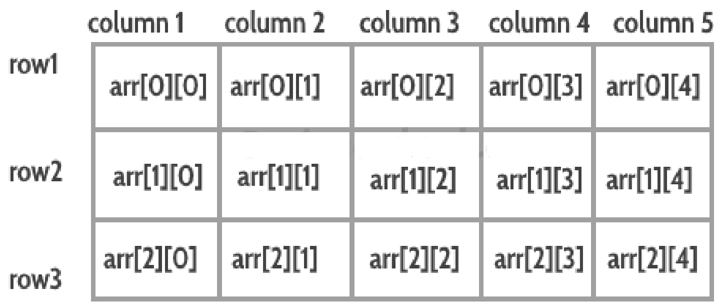
C programming language allows multidimensional arrays. Here is the general form of a multidimensional array declaration:

```
datatype name[size1][size2]...[sizeN];
```

For example, the following declaration creates a three dimensional 5 . 10 . 4 integer array:

```
int threedim[5][10][4];
```

Multidimensional arrays are also known as **array of arrays**. The data in multidimensional array is stored in a tabular form as shown in the diagram below:



	column 1	column 2	column 3	column 4	column 5
row1	arr[0][0]	arr[0][1]	arr[0][2]	arr[0][3]	arr[0][4]
row2	arr[1][0]	arr[1][1]	arr[1][2]	arr[1][3]	arr[1][4]
row3	arr[2][0]	arr[2][1]	arr[2][2]	arr[2][3]	arr[2][4]

Three dimensional arrays

Let's see how to declare, initialize and access Three Dimensional Array elements.

Declaring a three dimensional array:

int myarray[2][3][2];

Initialization:

We can initialize the array in many ways:

Method 1:

int arr[2][3][2] = {1, -1 ,2 ,-2 , 3 , -3, 4, -4, 5, -5, 6, -6};

Method 2:

This way of initializing is preferred as you can visualize the rows and columns here.

```
int arr[2][3][2] = {
    { {1,-1}, {2, -2}, {3, -3}},
    { {4, -4}, {5, -5}, {6, -6}}
}
```

Three dimensional array examples

```
#include <iostream>
using namespace std;
int main(){
  // initializing the array
  int arr[2][3][2] = {
    { {1,-1}, {2,-2}, {3,-3} },
    { {4,-4}, {5,-5}, {6,-6} }
  };
  // displaying array values
  for (int x = 0; x < 2; x++) {
    for (int y = 0; y < 3; y++) {
      for (int z = 0; z < 2; z++) {
        cout<<arr[x][y][z]<<" ";
      }
    }
  }
  return 0;
}
```

Output:

```
1 -1 2 -2 3 -3 4 -4 5 -5 6 -6
```

Command Line Argument

Command line argument is a parameter supplied to the program when it is invoked. Command line argument is an important concept in C programming. It is mostly used when you need to control your program from outside. Command line arguments are passed to the main() method.

Syntax: int main(int argc, char *argv[])

Here argc counts the number of arguments on the command line and argv[ ] is a pointer array which holds pointers of type char which points to the arguments passed to the program.

Example for Command Line Argument

```
#include <stdio.h>
#include <conio.h>
int main(int argc, char *argv[])
{
   int i;
   if( argc >= 2 )
   {
      printf("The arguments supplied are:\n");
      for(i = 1; i < argc; i++)
      {
         printf("%s\t", argv[i]);
      }
   }
   else
   {
      printf("argument list is empty.\n");
   }
   return 0;
}
```

Remember that argv[0] holds the name of the program and argv[1] points to the first command line argument and argv[n] gives the last argument. If no argument is supplied, argc will be 1.

Pointer to functions

It is possible to declare a pointer pointing to a function which can then be used as an argument in another function. A pointer to a function is declared as follows,

Syntax: type (*pointer-name)(parameter);

Example:

```
int (*sum)();   //legal declaration of pointer to function
int *sum();     //This is not a declaration of pointer to function
```

A function pointer can point to a specific function when it is assigned the name of that function.

```
int sum(int, int);
int (*s)(int, int);
s = sum;
```

Here s is a pointer to a function sum. Now sum can be called using function pointer s along with providing the required argument values.

```
s (10, 20);
```

Example of Pointer to Function

```
#include <stdio.h>
int sum(int x, int y)
```

```
{
   return x+y;
}

int main( )
{
   int (*fp)(int, int);
   fp = sum;
   int s = fp(10, 15);
   printf("Sum is %d", s);

   return 0;
}
```

Out Put

```
25
```

Complicated Function Pointer

You will find a lot of complex function pointer examples around, let's see one such example and try to understand it.

Syntax: void *(*foo) (int*);

It appears complex but it is very simple. In this case (*foo) is a pointer to the function, whose argument is of int* type and return type is void*.

Write a c program to print Array order reversal

```
#include<stdio.h>
main()
{
int i,a[100],c,k,j,n,num;
printf("Enter number of elements in array\n");
scanf("%d",&num);
printf("Enter numbers\n");
for(i=0;i<num;i++)
{
scanf("%d",&a[i]);

}
printf("Before arranging \n");
for(i=0;i<num;i++) printf("%d\n",a[i]);
k=num;
while(k>=0)
{
for(j=0;j<k-1;j++)
{
```

```
//using swap method c=a[j];
a[j]=a[j+1];
a[j+1]=c;

}
k--;
}
printf("After arranging \n");
for(i=0;i<num;i++) printf("%d\n",a[i]);
}
```

Write a c program to finding the maximum number in a set

```
#include<stdio.h>
void main()
{
int i,size,max=0,min=0;
printf("Enter size to find largest and smallest of given numbers\n");
scanf("%d",&size);
int a[size];
printf("Enter numbers in array\n");
for(i=0;i<size;i++)
{
scanf("%d",&a[i]);
}
min=a[0];
max=a[0];
for(i=0;i<size;i++)
{
if(a[i]>max)
{
max=a[i];
}
if(a[i]<min)
{
min=a[i];
}
}
printf("The largest number is %d\n",max);
printf("The smallest number is %d\n",min);
}
```

Write a c program to removal of duplicates from an order array

```
#include<stdio.h>
void removerep(int position,int a[],int size);
void main()
{
int i,j,size,count=0,k,position;
printf("Enter number of elements\n");
scanf("%d",&size);
int a[size],*temp;
printf("Enter numbers\n");
for(i=0;i<size;i++)
{
scanf("%d",&a[i]);
}
temp=a;
for(i=0;i<size;i++)
{
for(j=i+1;j<size;)
{
if(a[i]==a[j])
{
removerep(j,a,size);
size--;
}
else
{
j++;
}
}
}
printf("After removing repeated elements\n");
for(i=0;i<size;i++)
{
printf("%d\n",a[i]);
}
}
void removerep(int position,int a[],int size)
{
int i;
for(i=0;i<size;i++)//deleting repeated number in array
{
if(i<position)
{
a[i]=a[i];
}
if(i>=position)
{
a[i]=a[i+1];
}
}
}
```

Method II:

```
#include<stdio.h>
main()
{
int i,j,k,size,count=0,position;
printf("Enter number of elements\n");
scanf("%d",&size);
int a[size-1],*temp;
printf("Enter numbers\n");
for(i=0;i<size;i++)
{
scanf("%d",&a[i]);
}
temp=a;
for(i=0;i<size;i++)
{
for(j=i+1;j<size;)
{
if(a[i]==a[j])
{
for(k=j;k<size;k++)
{
a[k]=a[k+1];
}
size--;
}
else
{
j++;
}
}
}
printf("After removing repeated elements\n");
for(i=0;i<size;i++)
{
printf("%d\n",a[i]);
}
}
```

UNIT-V

UNIT-V

Sorting and Searching: Sorting by selection, sorting by exchange, sorting by insertion, sorting by partitioning, binary search.
Structures: Basics of structures, structures and functions, arrays of structures, pointers to structures, self-referential structures, table lookup, type def, unions, bit-fields.
Some other Features: Variable-length argument lists, formatted input-Scanf, file access, Error handling-stderr and exit, Line Input and Output, Miscellaneous Functions.

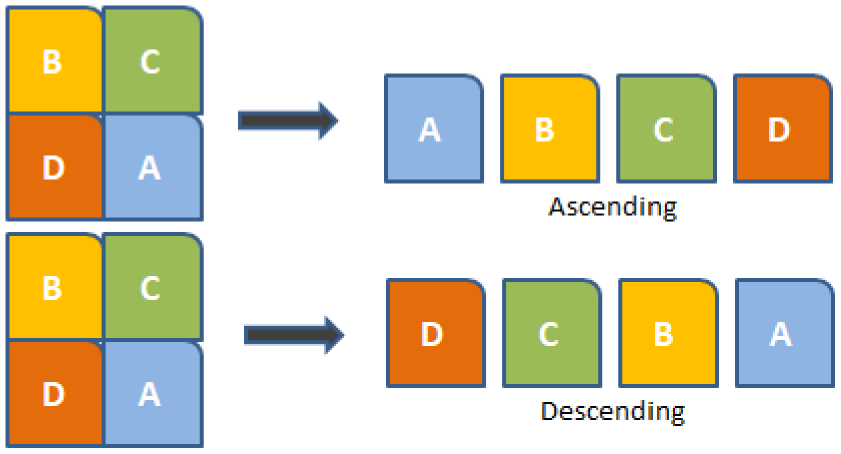
Sorting: arranges data in a sequence which makes searching easier.

Sorting is arranging the data in ascending or descending order. The term **sorting** came into picture, as humans realized the importance of searching quickly.

Sorting is a process of ordering individual elements of a list according to their proper rank, either in ascending or descending order. We can easily sort a list of elements by means of iteration /loop and if-condition check statements. Sorting algorithms can be implemented by any programming languages. One outer loop and one inner loop inside the outer loop will do the purpose.

There are so many things in our real life that we need to search for, like a particular record in database, roll numbers in merit list, a particular telephone number in telephone directory, a particular page in a book etc.

All this would have been a mess if the data was kept unordered and unsorted, but fortunately the concept of **sorting** came into existence, making it easier for everyone to arrange data in an order, hence making it easier to search.



Now suppose we have a record of elements inside a database having count of several thousands. The problem comes with the performance of the sorting. Computer takes much longer time to sort the elements in the list. Another problem comes when the size of individual elements are large and total number of elements are also large that the entire list of elements cannot be fitted in main memory for sorting at one go.

Therefore the sorting mechanism has been divided into are two main categories:-

Internal sorting
External sorting

Internal sorting: Internal sorting is done by loading all the elements in the main memory.

External sorting: When individual element size is more and number of elements are large enough to hold all the elements in main memory external sorting is used.

External sorting loads a portion of elements from secondary memory (like HDD) in main memory and sorts and then saves back to secondary storage. Later all the individual sorted fragments are merged in the main group.

We are discussing mainly internal sorting here. There are several algorithms used for this purpose each one has its own optimization techniques to execute the sort task in minimum CPU cycle thus to save time and energy.

The two main criteria's to judge which algorithm is better than the other have been:

1. Time taken to sort the given data. (Time Complexity)
2. Memory Space required to do so. (Space Complexity)

Different Sorting Algorithms

There are many different techniques available for sorting, differentiated by their efficiency and space requirements

- Bubble Sort
- Insertion Sort
- Selection Sort
- Quick Sort
- Merge Sort
- Heap Sort

Bubble Sort/Exchange Sort Algorithm

- Bubble Sort is a simple algorithm which is used to sort a given set of n elements provided in form of an array with n number of elements. Bubble Sort compares all the element one by one and sort them based on their values.

- If the given array has to be sorted in ascending order, then bubble sort will start by comparing the first element of the array with the second element, if the first element is greater than the second element, it will swap both the elements, and then move on to compare the second and the third element, and so on.
- If we have total n elements, then we need to repeat this process for n-1 times.

- It is known as bubble sort, because with every complete iteration the largest element in the given array, bubbles up towards the last place or the highest index, just like a water bubble rises up to the water surface.

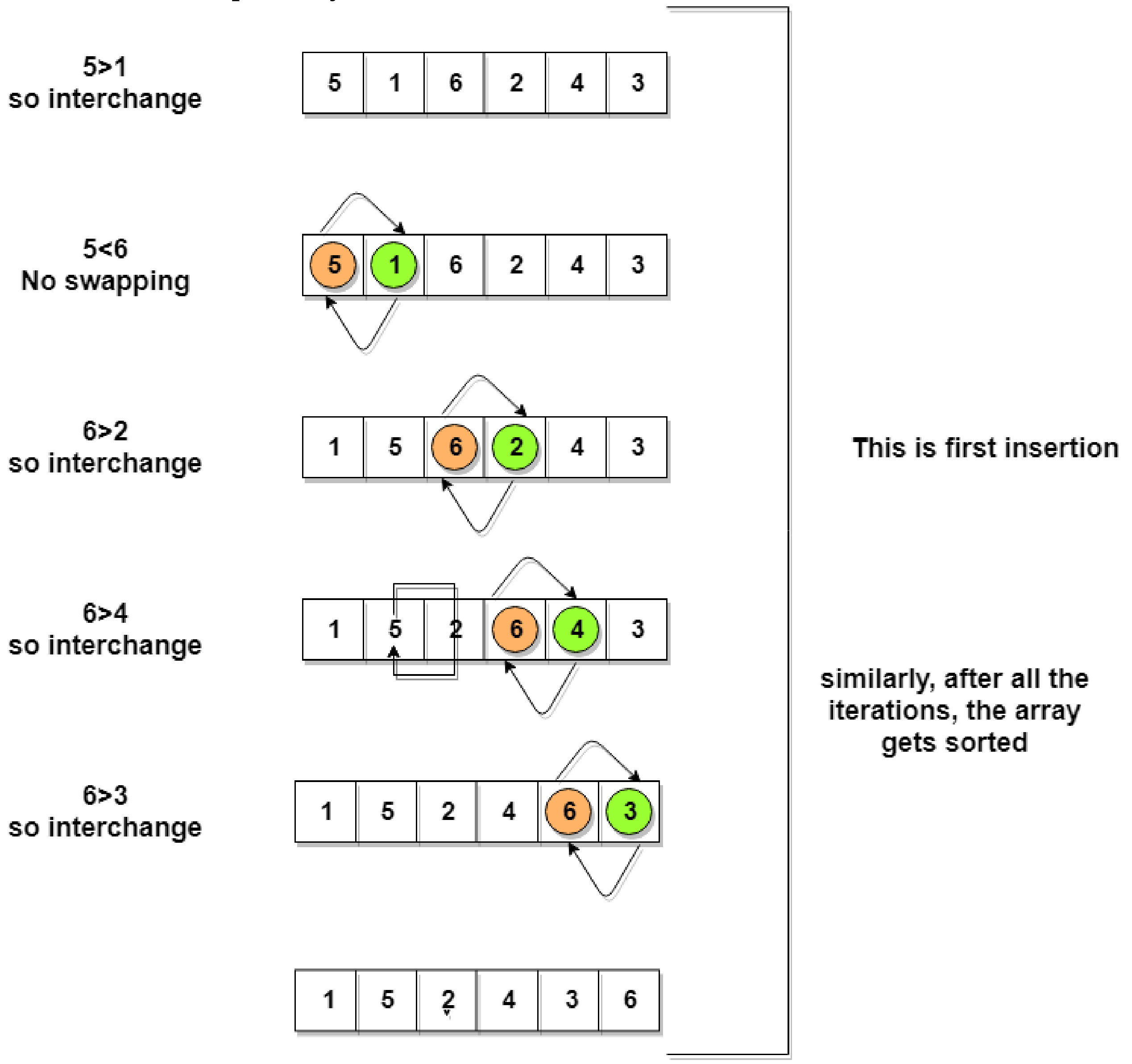
- Sorting takes place by stepping through all the elements one-by-one and comparing it with the adjacent element and swapping them if required.

Implementing Bubble Sort Algorithm

The following steps are involved in bubble sort(for sorting a given array in ascending order):

1. Starting with the first element (index = 0), compare the current element with the next element of the array.
2. If the current element is greater than the next element of the array, swap them.
3. If the current element is less than the next element, move to the next element. Repeat Step 1.

Let's consider an example array with values {5, 1, 6, 2, 4, 3}



So as we can see in the representation above, after the first iteration, 6 is placed at the last index, which is the correct position for it.
Similarly after the second iteration, 5 will be at the second last index, and so on.

Write a C program for bubble sort

```
#include <stdio.h>
void bubbleSort(int arr[], int n)
{
    int i, j, temp;
    for(i = 0; i < n; i++)
    {
        for(j = 0; j < n-i-1; j++)
        {
            if( arr[j] > arr[j+1])
            {
                // swap the elements
                temp = arr[j];
                arr[j] = arr[j+1];
                arr[j+1] = temp;
            }
        }
    }

    // print the sorted array
    printf("Sorted Array: ");
    for(i = 0; i < n; i++)
    {
        printf("%d  ", arr[i]);
    }
}

int main()
{
    int arr[100], i, n, step, temp;
    // ask user for number of elements to be sorted
    printf("Enter the number of elements to be sorted: ");
    scanf("%d", &n);
    // input elements if the array
    for(i = 0; i < n; i++)
    {
        printf("Enter element no. %d: ", i+1);
        scanf("%d", &arr[i]);
    }
    // call the function bubbleSort
    bubbleSort(arr, n);
    return 0;
}
```

Complexity Analysis of Bubble Sort

In Bubble Sort, n-1 comparisons will be done in the 1st pass, n-2 in 2nd pass, n-3 in 3rd pass and so on. So the total number of comparisons will be,

(n-1) + (n-2) + (n-3) + ... + 3 + 2 + 1
Sum = n(n-1)/2
i.e O(n2)

- ✓ Hence the time complexity of Bubble Sort is **O (n2).**
- ✓ The main advantage of Bubble Sort is the simplicity of the algorithm.
- ✓ The space complexity for Bubble Sort is **O (1),** because only a single additional memory space is required i.e. for temp variable.
- ✓ Also, the best case time complexity will be **O (n),** it is when the list is already sorted.
- ✓ Following are the Time and Space complexity for the Bubble Sort algorithm.

- ➢ Worst Case Time Complexity [Big-O]: **O(n2)**
- ➢ Best Case Time Complexity [Big-omega]: **O(n)**
- ➢ Average Time Complexity [Big-theta]: **O(n2)**
- ➢ Space Complexity: **O(1)**

Insertion Sort Algorithm

- ✓ Consider you have 10 cards out of a deck of cards in your hand. And they are sorted, or arranged in the ascending order of their numbers.

- ✓ If I give you another card, and ask you to insert the card in just the right position, so that the cards in your hand are still sorted. What will you do?

- ✓ Well, you will have to go through each card from the starting or the back and find the right position for the new card, comparing it's value with each card. Once you find the right position, you will insert the card there.

- ✓ Similarly, if more new cards are provided to you, you can easily repeat the same process and insert the new cards and keep the cards sorted too.

- ✓ This is exactly how insertion sort works. It starts from the index 1(not 0), and each index starting from index 1 is like a new card, that you have to place at the right position in the sorted sub-array on the left.

Characteristics of Insertion Sort:

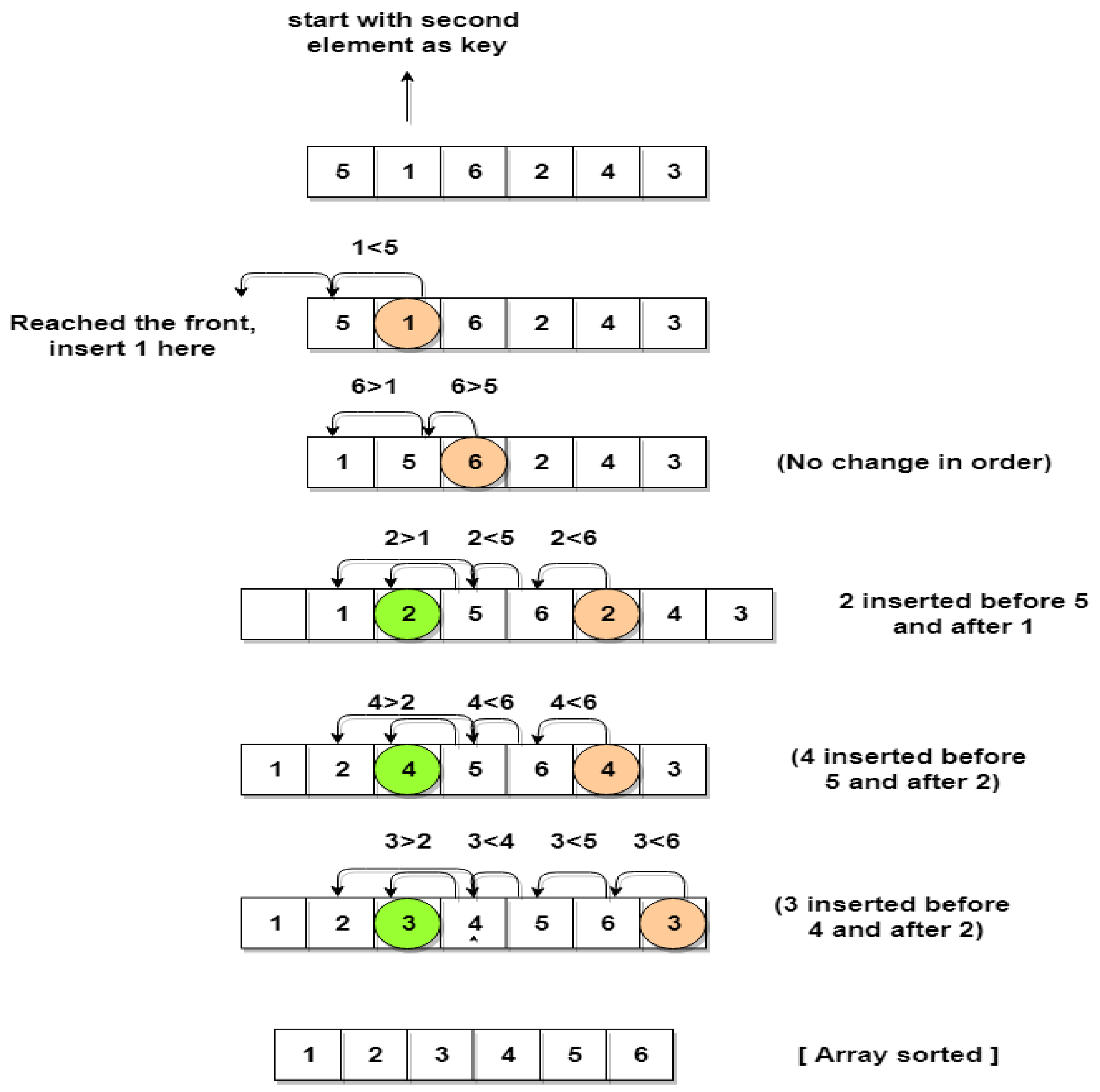
- ➢ It is efficient for smaller data sets, but very inefficient for larger lists.
- ➢ Insertion Sort is adaptive, that means it reduces its total number of steps if a partially sorted array is provided as input, making it efficient.
- ➢ It is better than Selection Sort and Bubble Sort algorithms.
- ➢ Its space complexity is less. Like bubble Sort, insertion sort also requires a single additional memory space.

- It is a stable sorting technique, as it does not change the relative order of elements which are equal.

How Insertion Sort Works?

- We start by making the second element of the given array, i.e. element at index 1, the key. The key element here is the new card that we need to add to our existing sorted set of cards
- We compare the key element with the element(s) before it, in this case, element at index 0:
- If the key element is less than the first element, we insert the key element before the first element.
- If the key element is greater than the first element, then we insert it after the first element.
- Then, we make the third element of the array as key and will compare it with elements to it's left and insert it at the right position.
- And we go on repeating this, until the array is sorted.

Let's consider Example array with values {5, 1, 6, 2, 4, 3}



Write a C program for insertion sort

```
#include<stdio.h>
int main()
{
  int i, j, count, temp, number[25];
  printf("How many numbers u are going to enter?: ");
  scanf("%d",&count);
  printf("Enter %d elements: ", count);
          // This loop would store the input numbers in array
  for(i=0;i<count;i++)
    scanf("%d",&number[i]);
          // Implementation of insertion sort algorithm
  for(i=1;i<count;i++)
{
    temp=number[i];
    j=i-1;
    while((temp<number[j])&&(j>=0))
{
   number[j+1]=number[j];
   j=j-1;
    }
    number[j+1]=temp;
  }
  printf("Order of Sorted elements: ");
  for(i=0;i<count;i++)
  printf(" %d",number[i]);
  return 0;
}
```

Complexity Analysis of Insertion Sort

As we mentioned above that insertion sort is an efficient sorting algorithm, as it does not run on preset conditions using for loops, but instead it uses one while loop, which avoids extra steps once the array gets sorted.

Even though insertion sort is efficient, still, if we provide an already sorted array to the insertion sort algorithm, it will still execute the outer for loop, thereby requiring n steps to sort an already sorted array of n elements, which makes its best case time complexity a linear function of n.

- Worst Case Time Complexity [Big-O]: **O(n2)**
- Best Case Time Complexity [Big-omega]: **O(n)**
- Average Time Complexity [Big-theta]: **O(n2)**
- Space Complexity: **O(1)**

Selection Sort Algorithm

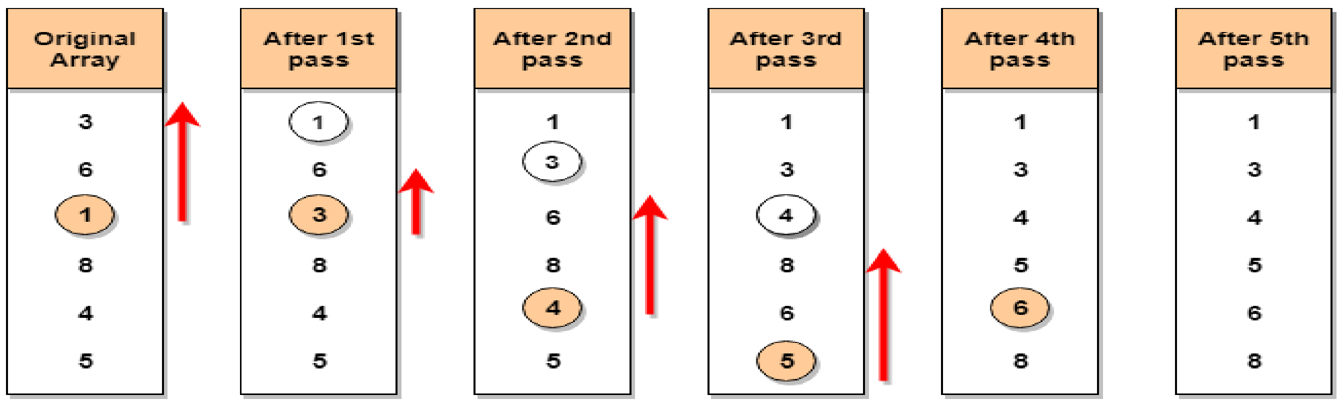
Selection sort is conceptually the simplest sorting algorithm. This algorithm will first find the smallest element in the array and swap it with the element in the first position, then it will find the second smallest element and swap it with the element in the second position, and it will keep on doing this until the entire array is sorted.

It is called selection sort because it repeatedly selects the next-smallest element and swaps it into the right place.

How Selection Sort Works?

- ✓ Starting from the first element, we search the smallest element in the array, and replace it with the element in the first position.
- ✓ We then move on to the second position, and look for smallest element present in the sub-array, starting from index 1, till the last index.
- ✓ We replace the element at the second position in the original array, or we can say at the first position in the sub-array, with the second smallest element.
- ✓ This is repeated, until the array is completely sorted.

Let's consider an Example array with values {3, 6, 1, 8, 4, 5}



- ✓ In the first pass, the smallest element will be 1, so it will be placed at the first position.
- ✓ Then leaving the first element, next smallest element will be searched, from the remaining elements. We will get 3 as the smallest, so it will be then placed at the second position.
- ✓ Then leaving 1 and 3(because they are at the correct position), we will search for the next smallest element from the rest of the elements and put it at third position and keep doing this until array is sorted.

Finding Smallest Element in a sub-array

- ✓ In selection sort, in the first step, we look for the smallest element in the array and replace it with the element at the first position. This seems doable, isn't it?

- ✓ Consider that you have an array with following values {3, 6, 1, 8, 4, 5}. Now as per selection sort, we will start from the first element and look for the smallest number in the array, which is 1 and we will find it at the index 2. Once the smallest number is found, it is swapped with the element at the first position.

- ✓ Well, in the next iteration, we will have to look for the second smallest number in the array. How can we find the second smallest number? This one is tricky?

- ✓ If you look closely, we already have the smallest number/element at the first position, which is the right position for it and we do not have to move it anywhere now. So we can say, that the first element is sorted, but the elements to the right, starting from index 1 are not.

- ✓ So, we will now look for the smallest element in the sub-array, starting from index 1, to the last index.

- ✓ After we have found the second smallest element and replaced it with element on index 1(which is the second position in the array), we will have the first two positions of the array sorted.

- ✓ Then we will work on the sub-array, starting from index 2 now, and again looking for the smallest element in this sub-array.

Implementing Selection Sort Algorithm

```
#include<stdio.h>
int main()
{
  int i, j, count, temp, number[25];
  printf("How many numbers u are going to enter?: ");
  scanf("%d",&count);
  printf("Enter %d elements: ", count);
        // Loop to get the elements stored in array
  for(i=0;i<count;i++)
    scanf("%d",&number[i]);
          // Logic of selection sort algorithm
  for(i=0;i<count;i++)
{
    for(j=i+1;j<count;j++)
{
      if(number[i]>number[j])
{
        temp=number[i];
        number[i]=number[j];
        number[j]=temp;
      }
    }
  }
  printf("Sorted elements: ");
  for(i=0;i<count;i++)
    printf(" %d",number[i]);
  return 0;
}
```

Note: Selection sort is an unstable sort i.e it might change the occurrence of two similar elements in the list while sorting. But it can also work as a stable sort when it is implemented using linked list.

Complexity Analysis of Selection Sort

- ✓ Selection Sort requires two nested for loops to complete itself, one for loop is in the function selection Sort, and inside the first loop we are making a call to another function indexOfMinimum, which has the second(inner) for loop.

- ✓ Hence for a given input size of n, following will be the time and space complexity for selection sort algorithm:

- ➢ Worst Case Time Complexity [Big-O]: **O(n2)**
- ➢ Best Case Time Complexity [Big-omega]: **O(n2)**
- ➢ Average Time Complexity [Big-theta]: **O(n2)**
- ➢ Space Complexity: **O(1)**

Quick Sort Algorithm /Sorting By Partitioning

Quick Sort is also based on the concept of **Divide and Conquer**, just like merge sort. But in quick sort all the heavy lifting(major work) is done while **dividing** the array into sub-arrays, while in case of merge sort, all the real work happens during **merging** the sub-arrays. In case of quick sort, the combine step does absolutely nothing.

It is also called **partition-exchange sort**. This algorithm divides the list into three main parts:

1. Elements less than the **Pivot** element
2. Pivot element(Central element)
3. Elements greater than the pivot element

Pivot element can be any element from the array, it can be the first element, the last element or any random element. In this tutorial, we will take the rightmost element or the last element as **pivot**.

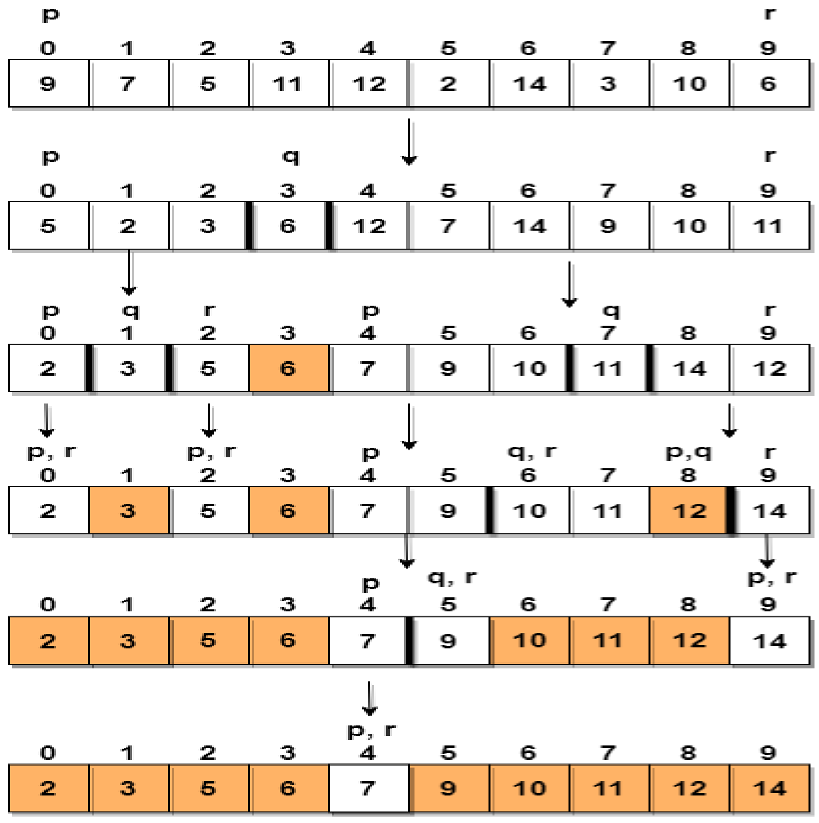
For example: In the array {52, 37, 63, 14, 17, 8, 6, 25}, we take **25** as **pivot**. So after the first pass, the list will be changed like this.{6 8 17 14 **25** 63 37 52}

Hence after the first pass, pivot will be set at its position, with all the elements **smaller** to it on its left and all the elements **larger** than to its right. Now 6 8 17 14 and 63 37 52 are considered as two separate sub-arrays, and same recursive logic will be applied on them, and we will keep doing this until the complete array is sorted

How Quick Sorting Works?

- ✓ After selecting an element as pivot, which is the last index of the array in our case, we divide the array for the first time.
- ✓ In quick sort, we call this partitioning. It is not simple breaking down of array into 2 sub-arrays, but in case of partitioning, the array elements are so positioned that all the elements smaller than the pivot will be on the left side of the pivot and all the elements greater than the pivot will be on the right side of it.
- ✓ And the pivot element will be at its final sorted position.
- ✓ The elements to the left and right may not be sorted.
- ✓ Then we pick sub-arrays, elements on the left of pivot and elements on the right of pivot, and we perform partitioning on them by choosing a pivot in the sub-arrays.

Let's consider an Example array with values {9, 7, 5, 11, 12, 2, 14, 3, 10, 6}



In step 1, we select the last element as the pivot, which is 6 in this case, and call for partitioning, hence re-arranging the array in such a way that 6 will be placed in its final position and to its left will be all the elements less than it and to its right, we will have all the elements greater than it.

Then we pick the sub-array on the left and the sub-array on the right and select a pivot for them, in the above diagram, we chose 3 as pivot for the left sub-array and 11 as pivot for the right sub-array.

And we again call for partitioning.

Write a c program to implement Quick sort algorithm

```
#include<stdio.h>
void quicksort(int number[25],int first,int last)
{
  int i, j, pivot, temp;
  if(first<last)
{
    pivot=first;
    i=first;
    j=last;
    while(i<j)
{
      while(number[i]<=number[pivot]&&i<last)
        i++;
      while(number[j]>number[pivot])
        j--;
      if(i<j)
{
        temp=number[i];
        number[i]=number[j];
        number[j]=temp;
      }
    }
    temp=number[pivot];
    number[pivot]=number[j];
    number[j]=temp;
    quicksort(number,first,j-1);
    quicksort(number,j+1,last);
  }
}

int main()
{
  int i, count, number[25];
  printf("How many elements are u going to enter?: ");
  scanf("%d",&count);
  printf("Enter %d elements: ", count);
  for(i=0;i<count;i++)
  scanf("%d",&number[i]);
  quicksort(number,0,count-1);
  printf("Order of Sorted elements: ");
  for(i=0;i<count;i++)
  printf(" %d",number[i]);
  return 0;
}
```

Complexity Analysis of Quick Sort

- ✓ For an array, in which partitioning leads to unbalanced sub-arrays, to an extent where on the left side there are no elements, with all the elements greater than the pivot, hence on the right side.

- ✓ And if keep on getting unbalanced sub-arrays, then the running time is the worst case, which is O(n2)

- ✓ Where as if partitioning leads to almost equal sub-arrays, then the running time is the best, with time complexity as O(n*log n).

- ➢ Worst Case Time Complexity [Big-O]: **O(n2)**
- ➢ Best Case Time Complexity [Big-omega]: **O(n*log n)**
- ➢ Average Time Complexity [Big-theta]: **O(n*log n)**
- ➢ Space Complexity: **O(n*log n)**

As we know now, that if sub-arrays partitioning produced after partitioning are unbalanced, quick sort will take more time to finish. If someone knows that you pick the last index as pivot all the time, they can intentionally provide you with array which will result in worst-case running time for quick sort.

To avoid this, you can pick random pivot element too. It won't make any difference in the algorithm, as all you need to do is, pick a random element from the array, swap it with element at the last index, make it the pivot and carry on with quick sort.

Space required by quick sort is very less, only **O (n*log n)** additional space is required.
Quick sort is not a stable sorting technique, so it might change the occurrence of two similar elements in the list while sorting.

Binary Search

- ✓ We start by comparing the element to be searched with the element in the middle of the list/array.
- ✓ If we get a match, we return the index of the middle element.
- ✓ If we do not get a match, we check whether the element to be searched is less or greater than in value than the middle element.
- ✓ If the element/number to be searched is greater in value than the middle number, then we pick the elements on the right side of the middle element (as the list/array is sorted, hence on the right, we will have all the numbers greater than the middle number), and start again from the step 1.
- ✓ If the element/number to be searched is lesser in value than the middle number, then we pick the elements on the left side of the middle element, and start again from the step 1.
- ✓ Binary Search is useful when there is large number of elements in an array and they are sorted.

- ✓ So a necessary condition for Binary search to work is that the list/array should be sorted.

Features of Binary Search

- It is great to search through large sorted arrays.
- It has a time complexity of O(log n) which is a very good time complexity.

Binary Search is applied on the sorted array or list of large size. It's time complexity of **O(log n)** makes it very fast as compared to other sorting algorithms. The only limitation is that the array or list of elements must be sorted for the binary search algorithm to work on it.

Implementing Binary Search Algorithm

- ✓ Start with the middle element:
- ✓ If the target value is equal to the middle element of the array, then return the index of the middle element.
- ✓ If not, then compare the middle element with the target value,
- ✓ If the target value is greater than the number in the middle index, then pick the elements to the right of the middle index, and start with Step 1.
- ✓ If the target value is less than the number in the middle index, then pick the elements to the left of the middle index, and start with Step 1.
- ✓ When a match is found, return the index of the element matched.
- ✓ If no match is found, then return -1

Write a c program search an element using binary search

```
#include <stdio.h>
void binary_search();
 int a[50], n, item, loc, beg, mid, end, i;
void main()
{
   printf("\nEnter size of an array: ");
   scanf("%d", &n);
   printf("\nEnter elements of an array in sorted form:\n");
   for(i=0; i<n; i++)
      scanf("%d", &a[i]);
   printf("\nEnter ITEM to be searched: ");
   scanf("%d", &item);
   binary_search();
   getch();
}
void binary_search()
{
   beg = 0;
   end = n-1;
   mid = (beg + end) / 2;
   while ((beg<=end) && (a[mid]!=item))
   {
      if (item < a[mid])
```

```
            end = mid - 1;
        else
            beg = mid + 1;
        mid = (beg + end) / 2;
    }
    if (a[mid] == item)
        printf("\n\nITEM found at location %d", mid+1);
    else
        printf("\n\nITEM doesn't exist");
}
```

OUTPUT:

```
Enter size of an array: 5
Enter elements of an array in sorted form:
10 25 35 64 89
Enter ITEM to be searched: 35
ITEM found at location 3
```

STRUCTURES

Arrays can be used to representing a group of interrelated data items of the same type under a common name. If we want to represent group data items of different data types under a common name, then the array is not useful in this case. For this purpose, “C” provides user defined data type known as structure.

Definition: Structure is a very important useful derived data type supported in c that allows grouping one or more variables of different data types with a single name.

A structure contains one or more data items of different type in which the individual element can differ in type. A simple structure may contain the integer element, float elements and character elements etc. the individual structure elements are called members.

struct keyword is used to define a structure. **struct** defines a new data type which is a collection of primary and derived data types.

Syntax:

```
struct [structure_tag]
{
   //member variable 1
   //member variable 2
   //member variable 3
   ...
}[structure_variables];
```

As you can see in the syntax above, we start with the **struct** keyword, then it's optional to provide your structure a name, we suggest you to give it a name, then inside the curly braces, we have to mention all the member variables, which are nothing but normal C language variables of different types like int, float, array etc.

After the closing curly brace, we can specify one or more structure variables, again this is optional.
Note: The closing curly brace in the structure type declaration must be followed by a semicolon **(;)**.

The general syntax of structure:

```
Struct <tagname>
        {
           datatype membername1;
            datatype membername2;
           datatype membername3;
           datatype membername4;
           ….
           …..
           ….
           datatype membername n;
        };
```

Example of Structure

```
struct Student
{
   char name[25];
   int age;
   char branch[10];
   // F for female and M for male
   char gender;
};
```

Here **struc**t Student declares a structure to hold the details of a student which consists of 4 data fields, namely name, age, branch and gender. These fields are called structure **elements or members**.

Each member can have different datatype, like in this case, name is an array of char type and age is of int type etc. Student is the name of the structure and is called as the structure tag.

Rules for declaring a structure:

- A structure must end with a semicolon.
- Element must be terminated.
- The structure variable must be accessed with structure variable with dot(.) operator

Structure variables can be declared in following two ways:

1) Declaring Structure variables separately

```
struct Student
{
   char name[25];
   int age;
   char branch[10];
   char gender;
};
struct Student S1, S2;      //declaring variables of struct Student
```

2) **Declaring Structure variables with structure definition**

```
          struct
          {
             char name[25];
             int age;
             char branch[10];
             char gender;
          }S1, S2;
```

Here S1 and S2 are variables of structure Student. However this approach is not much recommended.

Structure Initialization

Like a variable of any other datatype, structure variable can also be initialized at compile time.

```
   struct Patient
   {
      float height;
      int weight;
      int age;
   };
   struct Patient p1 = { 180.75 , 73, 23 };    //initialization

                              (or)
   struct Patient p1;
   p1.height = 180.75;     //initialization of each member separately
   p1.weight = 73;
p1.age = 23;
```

Accessing Structure Members

Structure members can be accessed and assigned values in a number of ways. Structure members have no meaning individually without the structure. In order to assign a value to any structure member, the member name must be linked with the structure variable using a **dot..**Operator also called period or member access operator.

For example:

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

struct Student
{
   char name[25];
   int age;
   char branch[10];
   char gender;
};

int main()
{
   struct Student s1;
```

```
            /* s1 is a variable of Student type and
                    age is a member of Student */
    s1.age = 18;
        /* using string function to add name  */

    strcpy(s1.name, "vemu");
            /* displaying the stored values*/

    printf("Name of Student 1: %s\n", s1.name);
    printf("Age of Student 1: %d\n", s1.age);

    return 0;
}
```

We can also use scanf() to give values to structure members through runtime.

```
        scanf(" %s ", &s1.name);
        scanf(" %d ", &s1.age);
```

Array of Structure

We can also declare an array of structure variables. in which each element of the array will represent a structure variable.

Example: struct employee emp[5];

```
#include<stdio.h>
struct Employee
{
   char ename[10];
   int sal;
};

struct Employee emp[5];
int i, j;
void ask()
{
   for(i = 0; i < 3; i++)
   {
      printf("\nEnter %dst Employee record:\n", i+1);
      printf("\nEmployee name:\t");
      scanf("%s", emp[i].ename);
      printf("\nEnter Salary:\t");
      scanf("%d", &emp[i].sal);
   }
   printf("\nDisplaying Employee record:\n");
   for(i = 0; i < 3; i++)
   {
      printf("\nEmployee name is %s", emp[i].ename);
      printf("\nSlary is %d", emp[i].sal);
   }
}
```

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

Nested Structures/Self-Referential Structures

Nesting of structures is also permitted in C language. Nested structures means, that one structure has another structure as member variable.

Example:

```
struct Student
{
    char[30] name;
    int age;
                /* here Address is a structure */
    struct Address
    {
        char[50] locality;
        char[50] city;
        int pincode;
    }addr;
};
```

Structure as Function Arguments

We can pass a structure as a function argument just like we pass any other variable or an array as a function argument.

Example:

```
#include<stdio.h>
struct Student
{
    char name[10];
    int roll;
};

void show(struct Student st);

void main()
{
    struct Student std;
    printf("\nEnter Student record:\n");
    printf("\nStudent name:\t");
    scanf("%s", std.name);
    printf("\nEnter Student rollno.:\t");
    scanf("%d", &std.roll);
    show(std);
}

void show(struct Student st)
{
        printf("\nstudent name is %s", st.name);
        printf("\nroll is %d", st.roll);
}
```

Pointer to Array of Structures

Like we have array of integers, array of pointers etc, we can also have array of structure variables. And to use the array of structure variables efficiently, we use pointers of structure type. We can also have pointer to a single structure variable, but it is mostly used when we are dealing with array of structure variables.

Example:

```
#include <stdio.h>

struct Book
{
    char name[10];
    int price;
}

int main()
{
    struct Book a;      //Single structure variable
    struct Book* ptr;   //Pointer of Structure type
    ptr = &a;

    struct Book b[10];  //Array of structure variables
    struct Book* p;     //Pointer of Structure type
    p = &b;

    return 0;
}
```

Accessing Structure Members with Pointer

To access members of structure using the structure variable, we used the **dot.Operator**. But when we have a pointer of structure type, we use arrow → to access structure members.

Example:

```
#include <stdio.h>

struct my_structure
 {
    char name[20];
    int number;
    int rank;
};
int main()
{
    struct my_structure variable = {"VEMU", 35, 1};

    struct my_structure *ptr;
    ptr = &variable;

    printf("NAME: %s\n", ptr->name);
    printf("NUMBER: %d\n", ptr->number);
    printf("RANK: %d", ptr->rank);

    return 0;
}
```

TYPEDEF

Typedef is a keyword used in C language to assign alternative names to existing datatypes. Its mostly used with user defined datatypes, when names of the datatypes become slightly complicated to use in programs.

syntax for typedef:

```
                typedef <existing_name><alias_name>
```

Example for how typedef actually works.

```
          typedef unsigned long ulong;
```

The above statement define a term ulong for an unsigned long datatype. Now this ulong identifier can be used to define unsigned long type variables.

```
                ulong i, j;
```

Application of typedef

typedef can be used to give a name to user defined data type as well. Lets see its use with structures.

```
          typedef struct
          {
             type member1;
             type member2;
             type member3;
          } type_name;
```

Here type_name represents the stucture definition associated with it. Now this type_name can be used to declare a variable of this stucture type.

```
          type_name t1, t2;
```

Structure definition using typedef

Let's take a simple code example to understand how we can define a structure in C using typedef keyword.

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

typedef struct employee
{
   char name[50];
   int salary;
}emp;

void main( )
{
   emp e1;
   printf("\nEnter Employee record:\n");
   printf("\nEmployee name:\t");
   scanf("%s", e1.name);
   printf("\nEnter Employee salary: \t");
   scanf("%d", &e1.salary);
   printf("\nstudent name is %s", e1.name);
   printf("\nroll is %d", e1.salary);
}
```

typedef and Pointers

typedef can be used to give an alias name to pointers also. Here we have a case in which use of typedef is beneficial during pointer declaration.

In Pointers * binds to the right and not on the left.

```
int* x, y;
```

By this declaration statement, we are actually declaring **x** as a pointer of type int, whereas **y** will be declared as a plain int variable.

```
typedef int* IntPtr;
IntPtr x, y, z;
```

But if we use typedef like we have used in the example above, we can declare any number of pointers in a single statement.

Bitfields:

Bitfields is a concept supported in programming language C that allows the programmer to allocate **bits** of memory to variables instead of **bytes** of memory.

If a variable needs store a value in between 0 and 7 then no need to allocate bytes of memory only bits of memory is sufficient and this can be done by accompanying the variable with colon(**:**) followed by non-negative integer value.

This concept can be applied only to members of a structure.

```
struct abc
{
  unsigned int a;
unsigned int b;
     };
```

This structure requires 8 bytes of memory space but in actual we are going to store either 0 or 1 in each of the variables. If we use such variables inside a structure then we can define the width of a variable which specifies that we are going to use only those number of bytes. For example, above structure can be modified as follows:

```
struct abc
{
  unsigned int a : 1;
  unsigned int b : 1;
};
```

Now, the above structure will require 4 bytes of memory space for status variable but only 2 bits will be used to store the values. If you will use up to 32 variables each one with a width of 1 bit , then also status structure will use 4 bytes, but as soon as you will have 33 variables, then it will allocate next slot of the memory and it will start using 8 bytes.

For example:

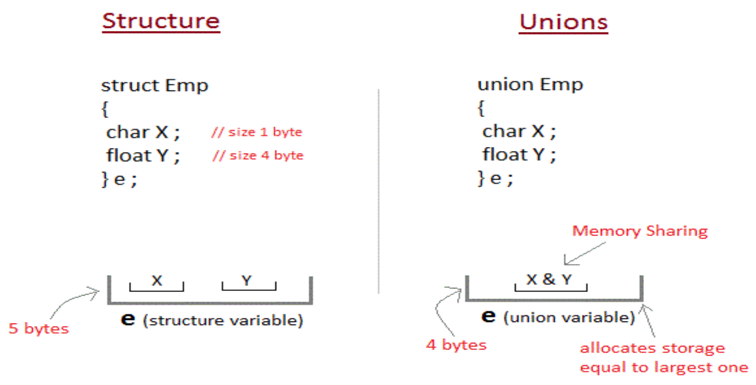
```
struct abc1
{
  int a;
  int b;
} ;

struct abc2
{
  int a : 1;
  int b : 1;
} ;

int main( )
{
   printf( "Memory size occupied by abc1 : %zu\n", sizeof(struct abc1));
   printf( "Memory size occupied by abc2 : %zu\n", sizeof(struct abc2));
   return 0;
}
```

Output :

```
Memory size occupied by abc1 : 8
Memory size occupied by abc2 : 4
```

Unions

Unions are conceptually similar to structures. The syntax to declare/define a union is also similar to that of a structure. The only difference is in terms of storage. In structure each member has its own storage location, whereas all members of union use a single shared memory location which is equal to the size of its largest data member.



This implies that although a union may contain many members of different types, it cannot handle all the members at the same time. A union is declared using the **union** keyword.

Syntax:

```
union item
{
   int m;
   float x;
   char c;
}It1;
```

This declares a variable **It1** of type union item. This union contains three members each with a different data type. However only one of them can be used at a time. This is due to the fact that only one location is allocated for all the union variables, irrespective of their size. The compiler allocates the storage that is large enough to hold the largest variable type in the union.
In the union declared above the member **x** requires 4 bytes which is largest amongst the members for a 16-bit machine. Other members of union will share the same memory address.

Accessing a Union Member

Syntax for accessing any union member is similar to accessing structure members,

```
union test
{
   int a;
   float b;
   char c;
}t;
```

```
t.a;    //to access members of union t
t.b;
t.c;
```

Time for an Example

```
#include <stdio.h>
union item
{
   int a;
   float b;
   char ch;
};
int main( )
{
   union item it;
   it.a = 12;
   it.b = 20.2;
   it.ch = 'z';
   printf("%d\n", it.a);
   printf("%f\n", it.b);
   printf("%c\n", it.ch);
   return 0;
}
```

Output:

```
        -26426
        20.1999
        Z
```

As you can see here, the values of **a** and **b** get corrupted and only variable c prints the expected result. This is because in union, the memory is shared among different data types. Hence, the only member whose value is currently stored will have the memory.

In the above example, value of the variable **c** was stored at last, hence the value of other variables is lost.

ENUMERATION

Enumeration is a user defined datatype in C language. It is used to assign names to the integral constants which makes a program easy to read and maintain. The keyword "enum" is used to declare an enumeration.

Here is the syntax of enum,

```
                enum enum_name{const1, const2,  ....... };
```

The enum keyword is also used to define the variables of enum type. There are two ways to define the variables of enum type as follows.

```
enum week{sunday, monday, tuesday, wednesday, thursday, friday, saturday};
enum week day;
```

Example

```
#include<stdio.h>
enum week{Mon=10, Tue, Wed, Thur, Fri=10, Sat=16, Sun};
enum day{Mond, Tues, Wedn, Thurs, Frid=18, Satu=11, Sund};
int main() {
printf("The value of enum week: %d\t%d\t%d\t%d\t%d\t%d\t%d\n\n",Mon , Tue, Wed, Thur, Fri, Sat, Sun);
printf("The default value of enum day: %d\t%d\t%d\t%d\t%d\t%d\t%d",Mond , Tues, Wedn, Thurs, Frid, Satu,
Sund);
   return 0;
}
```

Output

```
The value of enum week: 10  11      12      13      10      16      17
The default value of enum day: 0        1       2       3       18      11      12
```

In the above program, two enums are declared as week and day outside the main() function. In the main() function, the values of enum elements are printed.

```
enum week{Mon=10, Tue, Wed, Thur, Fri=10, Sat=16, Sun};
enum day{Mond, Tues, Wedn, Thurs, Frid=18, Satu=11, Sund};
int main()
 {
   printf("The value of enum week: %d\t%d\t%d\t%d\t%d\t%d\t%d\n\n",Mon , Tue, Wed, Thur, Fri,
Sat, Sun);
   printf("The default value of enum day: %d\t%d\t%d\t%d\t%d\t%d\t%d",Mond , Tues, Wedn, Thurs,
Frid, Satu, Sund);
}
```

VARIABLE LENGTH ARGUMENT

Variable length argument is a feature that allows a function to receive any number of arguments. There are situations where we want a function to handle variable number of arguments according to requirement.
1) Sum of given numbers.
2) Minimum of given numbers.
and many more.

- Variable length of arguments is represented by three dotes (…)
- **stdarg.h** header file should be included to make use of variable length argument functions.

```
#include <stdio.h>
#include <stdarg.h>
 int add(int num,...);
 int main()
{
    printf("The value from first function call = " \  "%d\n", add(2,2,3));
    printf("The value from second function call= " \ "%d \n", add(4,2,3,4,5));

    /*Note - In function add(2,2,3),
                first 2 is total number of arguments
                2,3 are variable length arguments
             In function add(4,2,3,4,5),
                4 is total number of arguments
                2,3,4,5 are variable length arguments
    */
     return 0;
}

int add(int num,...)
{
    va_list valist;
    int sum = 0;
    int i;
     va_start(valist, num);
    for (i = 0; i < num; i++)
    {
       sum += va_arg(valist, int);
    }
    va_end(valist);
    return sum;
}
```

OUTPUT:

```
        The value from first function call = 5
        The value from second function call= 14
```

In the above program, function “add” is called twice. But, number of arguments passed to the function gets varies for each. So, 3 dots (…) are mentioned for function ‘add” that indicates that this function will get any number of arguments at run time.

WHAT IS FILE?

File is a collection of bytes that is stored on secondary storage devices like disk. There are two kinds of files in a system. They are,

- **Text files (ASCII)**
- **Binary files**

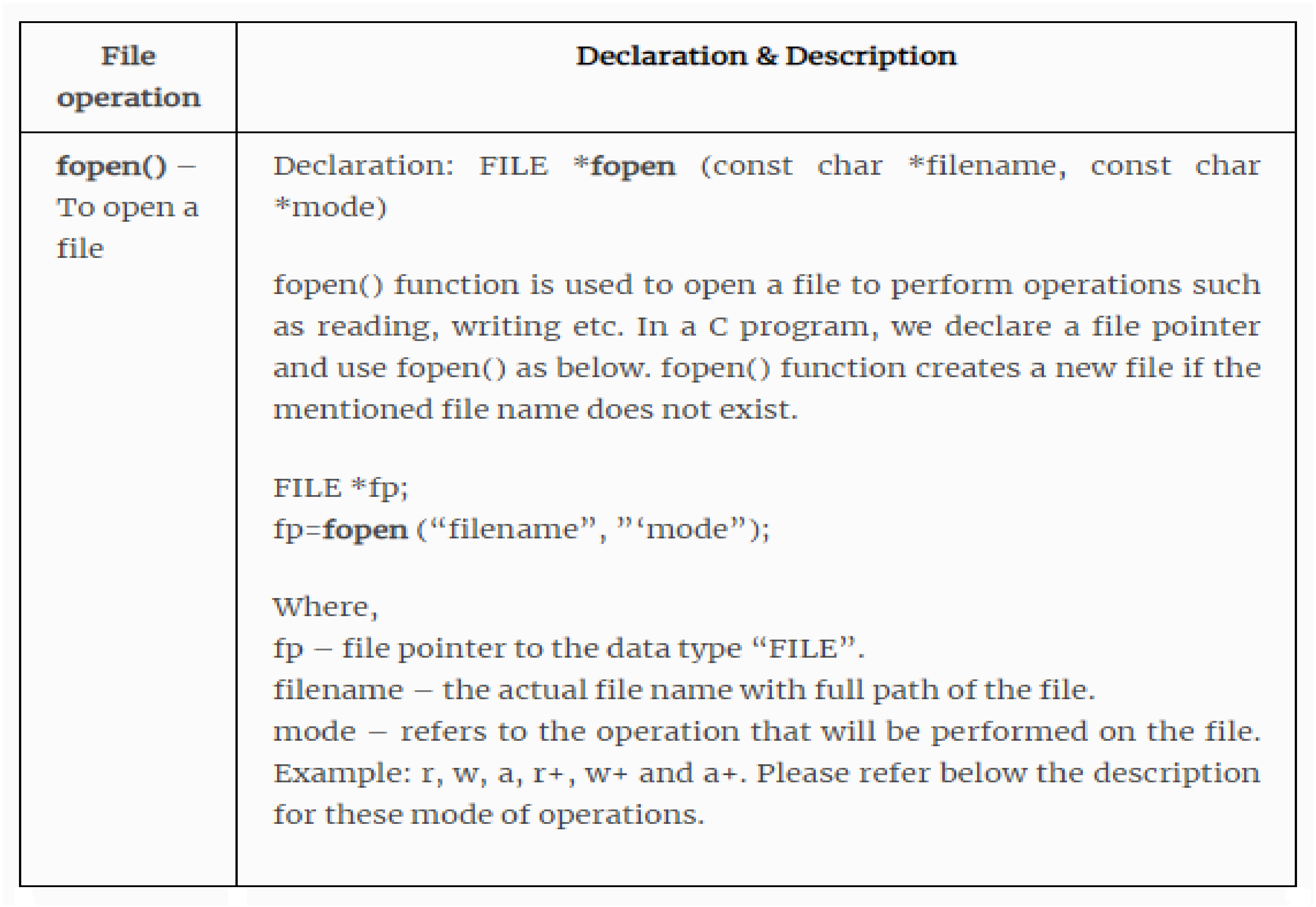
✓ Text files contain ASCII codes of digits, alphabetic and symbols.
✓ Binary file contains collection of bytes (0’s and 1’s). Binary files are compiled version of text files.

BASIC FILE OPERATIONS IN C PROGRAMMING:

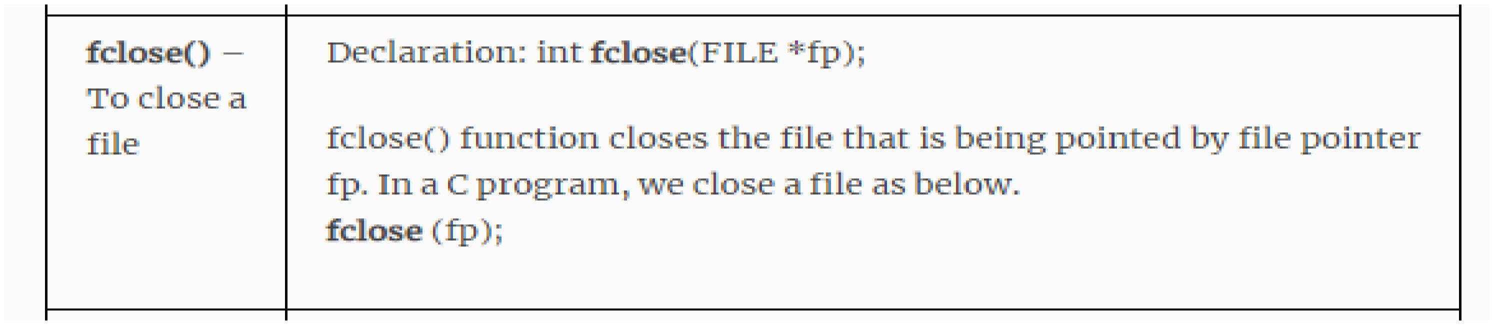
There are 4 basic operations that can be performed on any files in C programming language. They are,

- Opening/Creating a file
- Closing a file
- Reading a file
- Writing in a file

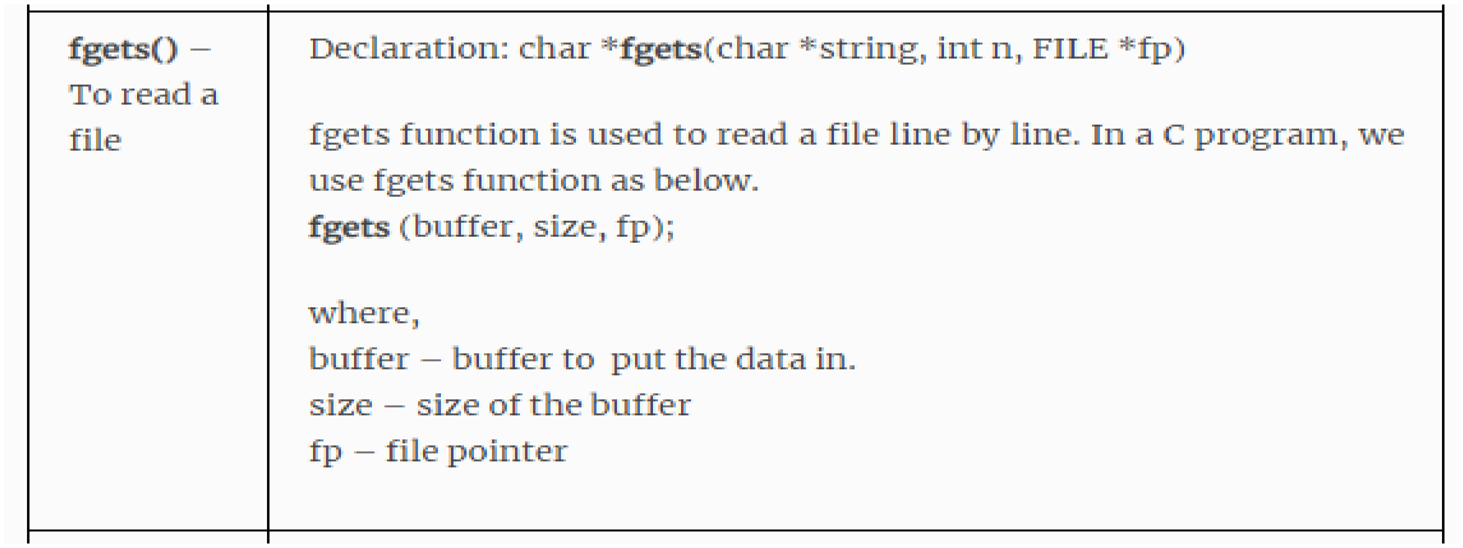
Let us see the syntax for each of the above operations in a table:



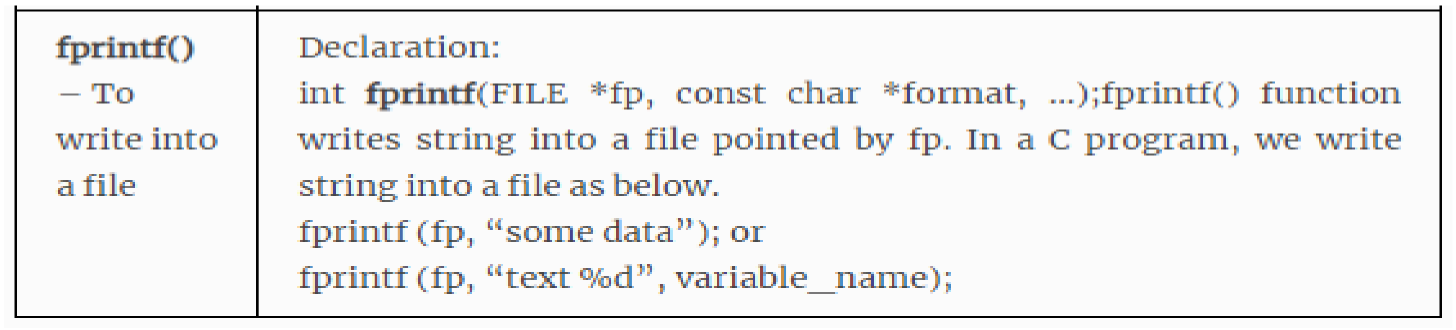
File operation	Declaration & Description
fopen() – To open a file	Declaration: FILE ***fopen** (const char *filename, const char *mode) fopen() function is used to open a file to perform operations such as reading, writing etc. In a C program, we declare a file pointer and use fopen() as below. fopen() function creates a new file if the mentioned file name does not exist. FILE *fp; fp=**fopen** (“filename”, ”‘mode”); Where, fp – file pointer to the data type “FILE”. filename – the actual file name with full path of the file. mode – refers to the operation that will be performed on the file. Example: r, w, a, r+, w+ and a+. Please refer below the description for these mode of operations.



fclose() – To close a file	Declaration: int **fclose**(FILE *fp);
	
	fclose() function closes the file that is being pointed by file pointer fp. In a C program, we close a file as below.
	fclose (fp);



fgets() – To read a file	Declaration: char ***fgets**(char *string, int n, FILE *fp) fgets function is used to read a file line by line. In a C program, we use fgets function as below. **fgets** (buffer, size, fp); where, buffer – buffer to put the data in. size – size of the buffer fp – file pointer



fprintf() – To write into a file	Declaration: int **fprintf**(FILE *fp, const char *format, ...);fprintf() function writes string into a file pointed by fp. In a C program, we write string into a file as below. fprintf (fp, "some data"); or fprintf (fp, "text %d", variable_name);

- r – Opens a file in read mode and sets pointer to the first character in the file. It returns null if file does not exist.
- w – Opens a file in write mode. It returns null if file could not be opened. If file exists, data are overwritten.
- a – Opens a file in append mode. It returns null if file couldn't be opened.
- r+ – Opens a file for read and write mode and sets pointer to the first character in the file.
- w+ – opens a file for read and write mode and sets pointer to the first character in the file.
- a+ – Opens a file for read and write mode and sets pointer to the first character in the file. But, it can't modify existing contents.

EXAMPLE PROGRAM FOR FILE OPEN, FILE WRITE AND FILE CLOSE

```
/ * Open, write and close a file : */
# include <stdio.h>
# include <string.h>
```

```
int main( )
{
   FILE *fp ;
   char data[50];
   // opening an existing file
   printf( "Opening the file test.c in write mode" ) ;
   fp = fopen("test.c", "w") ;
   if ( fp == NULL )
   {
      printf( "Could not open file test.c" ) ;
      return 1;
   }
   printf( "\n Enter some text from keyboard” \
          “ to write in the file test.c" ) ;
   // getting input from user
   while ( strlen ( gets( data ) ) > 0 )
   {
      // writing in the file
      fputs(data, fp) ;
      fputs("\n", fp) ;
   }
   // closing the file
   printf("Closing the file test.c") ;
   fclose(fp) ;
   return 0;
}
```

OUTPUT:

```
Opening the file test.c in write mode
Enter some text from keyboard to write in the file test.c
Hai, How are you?
Closing the file test.c
```

File Input/Output

A file represents a sequence of bytes on the disk where a group of related data is stored. File is created for permanent storage of data. It is a ready made structure.

In C language, we use a structure pointer of file type to declare a file.

FILE *fp;

C provides a number of functions that helps to perform basic file operations. Following are the functions.

Function	**description**
fopen()	create a new file or open a existing file
fclose()	closes a file
getc()	reads a character from a file
putc()	writes a character to a file
fscanf()	reads a set of data from a file
fprintf()	writes a set of data to a file
getw()	reads a integer from a file
putw()	writes a integer to a file
fseek()	set the position to desire point
ftell()	gives current position in the file
rewind()	set the position to the begining point

Opening a File or Creating a File

The fopen() function is used to create a new file or to open an existing file.

General Syntax:

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

Here, *fp is the FILE pointer (FILE *fp), which will hold the reference to the opened(or created) file. filename is the name of the file to be opened and mode specifies the purpose of opening the file. Mode can be of following types,

mode	description
r	opens a text file in reading mode
w	opens or create a text file in writing mode.
a	opens a text file in append mode
r+	opens a text file in both reading and writing mode
w+	opens a text file in both reading and writing mode
a+	opens a text file in both reading and writing mode
rb	opens a binary file in reading mode
wb	opens or create a binary file in writing mode
ab	opens a binary file in append mode
rb+	opens a binary file in both reading and writing mode
wb+	opens a binary file in both reading and writing mode
ab+	opens a binary file in both reading and writing mode

Closing a File
The fclose() function is used to close an already opened file.

General Syntax :

```
int fclose( FILE *fp);
```

Here fclose() function closes the file and returns zero on success, or EOF if there is an error in closing the file. This EOF is a constant defined in the header file stdio.h.

Input/Output operation on File

In the above table we have discussed about various file I/O functions to perform reading and writing on file. getc() and putc() are the simplest functions which can be used to read and write individual characters to a file.

```
#include<stdio.h>
int main()
{
   FILE *fp;
   char ch;
   fp = fopen("one.txt", "w");
   printf("Enter data...");
   while( (ch = getchar()) != EOF) {
      putc(ch, fp);
   }
   fclose(fp);
   fp = fopen("one.txt", "r");

   while( (ch = getc(fp)! = EOF)
   printf("%c",ch);

   // closing the file pointer
   fclose(fp);
      return 0;
}
```

Reading and Writing to File using fprintf() and fscanf()

```
#include<stdio.h>
struct emp
{
   char name[10];
   int age;
};

void main()
{
   struct emp e;
   FILE *p,*q;
   p = fopen("one.txt", "a");
   q = fopen("one.txt", "r");
   printf("Enter Name and Age:");
   scanf("%s %d", e.name, &e.age);
   fprintf(p,"%s %d", e.name, e.age);
   fclose(p);
   do
   {
       fscanf(q,"%s %d", e.name, e.age);
       printf("%s %d", e.name, e.age);
   }
   while(!feof(q));
}
```

- In this program, we have created two FILE pointers and both are refering to the same file but in different modes.
- fprintf() function directly writes into the file, while fscanf() reads from the file, which can then be printed on the console using standard printf() function.

Difference between Append and Write Mode

Write (w) mode and Append (a) mode, while opening a file are almost the same. Both are used to write in a file. In both the modes, new file is created if it doesn't exists already.

The only difference they have is, when you open a file in the write mode, the file is reset, resulting in deletion of any data already present in the file. While in append mode this will not happen. Append mode is used to append or add data to the existing data of file(if any). Hence, when you open a file in Append(a) mode, the cursor is positioned at the end of the present data in the file.

Reading and Writing in a Binary File

A Binary file is similar to a text file, but it contains only large numerical data. The Opening modes are mentioned in the table for opening modes above.

fread() and fwrite() functions are used to read and write is a binary file.

fwrite(data-element-to-be-written, size_of_elements, number_of_elements, pointer-to-file);

fread() is also used in the same way, with the same arguments like fwrite() function. Below mentioned is a simple example of writing into a binary file

```
const char *mytext = "The quick brown fox jumps over the lazy dog";
FILE *bfp= fopen("test.txt", "wb");
if (bfp)
{
   fwrite(mytext, sizeof(char), strlen(mytext), bfp);
   fclose(bfp);
}
```

fseek(), ftell() and rewind() functions

fseek(): It is used to move the reading control to different positions using fseek function.

ftell(): It tells the byte location of current position of cursor in file pointer.

rewind(): It moves the control to beginning of the file.

Table lookup:-

An array or matrix of data that contains items that are searched. Lookup tables may be arranged as key-value pairs, where the keys are the data items being searched (looked up) and the values are either the actual data or pointers to where the data are located. Sometimes, lookup tables contain only data items (just values, not key-value pairs). For example, in a 256-color palette, the location in the table and the key are synonymous (item 0, item 1, item 2, etc).

Example:

```
#define TABLE_SIZE (27u)
uint8_t lookup(uint8_t index)
{
uint16_t value;
static const __flash uint16_t lookup[TABLE_SIZE] =
{
946u, 2786u, ... 89u
};
if (index < TABLE_SIZE)
{
value = lookup[index];
}
else
{
//Handle error
}
...
}
```