ANNAMACHARYA UNIVERSITY

(ESTD UNDER AP PRIVATE UNIVERSITIES (ESTABLISHMENT AND REGULATION) ACT, 2016

New Boyanapalli, Rajampet,
Annamayya (Dist), Andhra Pradesh – 516 126

DEPARTMENT OF ELECTRICAL & ELECTRONICS ENGINEERING

Course: Computational Problem Solving

Course Code: **24ACSE11T**

Branch:EEE

Prepared by: P.Bhaskara Prasad

Designation:Assistant Professor

Department: EEE

UNIT-I

Overview of Computers and Programming - Electronic Computers then and Now, Computer Hardware, Computer Software, Algorithm, Flowcharts, Software Development Method, Applying the Software Development Method.

Types, Operators and Expressions: variable names-Data Types and Sizes-constants-Declarations-Arithmetic Operator-Relational and Logical Operators-Type conversions-Increment and Decrement Operators-Bitwise Operators-Assignment Operators and Expressions-Conditional Expressions-precedence and Order of Evaluation.

---

1. Overview of computer

The word "computer" comes from the word "compute", which means to calculate. Hence, a computer is normally considered to be calculating device, which can perform arithmetic (addition, subtraction…, etc) operations at enormous speed.

Computer is a fast electronic calculating machine that accepts digitized information from the user, processes it according to a sequence of instructions stored in the internal storage, and provides the processed information to the user.

Sequence of instructions are stored in the internal storage is called *computer program.* And internal storage is called *computer memory.*

The cycle of operation of computer is known as the INPUT – PROCESS – OUTPUT and is shown as following figure.

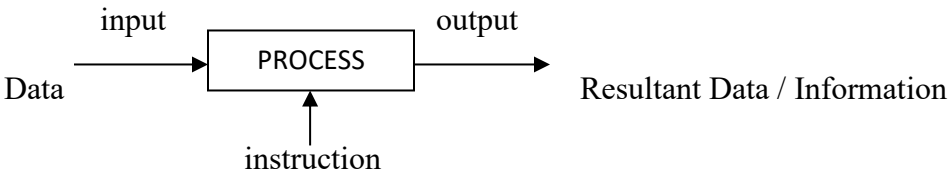


Fig: 1. Operation of computer

Data is nothing but raw fact (alphabets, numbers, Special Characters etc.). For example: a, b, c, @ etc. Information is nothing but meaningful data (resultant data). For example: c=a+b.

Computer should be instructed to perform certain task. Computer on its own cannot think or perform any task. Computer is a machine that simply follows the instructions given to it.

Computer is an electronic calculating machine in which all information is indicated by one of two states. The two states are binary 1 and 0 (binary digits).

1.1. Characteristics of computer

The increasing popularity of computer has proved that it is very powerful and useful tool. The power and usefulness of this popular tool are mainly due to its following characteristics:

- ✓ Automatic
- ✓ Speed
- ✓ Accuracy
- ✓ Diligence
- ✓ Versatility
- ✓ Storage
- ✓ Reliability

1.1.1 Automatic

A machine is said to be automatic, if it works by itself without human interaction. Computers are automatic machines because once started on a job, they carry on, until the job is finished, normally without any human assistance.

However, computer being machines cannot start them. They cannot go out and find their own problems and solutions. They have instructed. That is, a computer works from a program of coded instruction, which specify exactly how a particular job is to be done.

1.1.2 Speed

A computer is a very fast device. It can perform in a few seconds, the amount of work that a human being can do in an entire year, if he worked day and night. The speed of the computer can be measured in *milliseconds (10^{-3}), microseconds (10^{-6}), nanoseconds (10^{-9}),* and *picoseconds (10^{-12}).*

1.1.3 Accuracy

The accuracy of computer is consistently high and degree of accuracy of a particular computer depends upon its design. When human works for a few hours continuously, he may commit some mistakes but even it computer working for a long time with high speed, it can't commit mistakes or errors.

Error can occur in a computer; however, these are mainly due to human rather than technological weakness. For example: error may occur due to imprecise thinking by the programmer (a person who writes instructions for a computer to solve a particular problem) or incorrect input data.

1.1.4 Diligence

1.1.5 Unlike human being, a computer is free from monotony, tiredness and lack of concentration. It can continuously work for hours, without creating any errors and without grumbling. This characteristic is especially useful for those jobs where same tasks are done again and again.

1.1.6 Versatility

Computers are quiet versatile in nature. It can perform multiple tasks simultaneously. The computer can communicate with other computer in a net work by sending and receiving data in various forms like text, video, image, sound, graphics etc.

1.1.7 Storage

Computer has the ability to store large amount of data and retrieve the information that the user wants in a few seconds. All the data can be stored in memory the capacity or size of memory can be measured in terms of *bits* and *bytes*.

1.1.8 Reliability

The term reliability refers to the ability of computer related hardware or software components to consistently perform according to its specifications. Reliability is measurement and performance of computer.

1.2 Block Diagram of the computer (or) functional units

The computer consists of five functionally independent units:

- ✓ Input unit
- ✓ Memory unit
- ✓ Arithmetic & logic unit
- ✓ Control unit
- ✓ Output unit

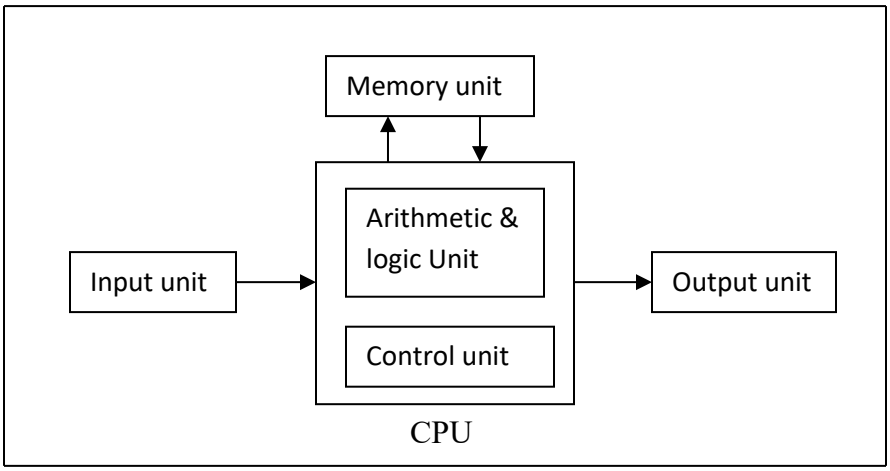


Fig 2: Block Diagram of Computer

The Fig 2: shows these five functional units of a computer and its physical units of a computer and its physical locations in the computer.

The input unit accepts the digital information from the user with the help of input unit devices such as keyboard, mouse, microphone, etc. the information received from the input unit is either stored in the memory for later use or immediately used by the arithmetic and logical unit to perform the desired operations.

The program stored in the memory decides the processing steps and the processed output sent to the user with the help of output devices or it is stored in the memory for later reference. All the above maintained activities are coordinated and controlled by the control unit.

The arithmetic and logic unit in conjunction with control unit is commonly called central processing unit (CPU).

2. Electronic Computers Then and Now

2.1 Early computers

Some of the well known early computers

- ✓ Blaise Pascal
- ✓ Speedo meter
- ✓ Analytical engine
- ✓ Atanas off-berry computer
- ✓ ENIAC
- ✓ EDSAC
- ✓ UNIVAC

2.1.1 Blaise Pascal

The first mechanical adding machine was invented by Blaise Pascal in1642. Later, in the year 1671 Baron Gottfried of Germany invented the first calculator for multiplication.

2.1.2 Speedo meter

It performs arithmetic operations and logarithms. by the use of logarithms, arithmetic operations like multiplications, divisions, Square roots and fractions can be calculated easily.

2.1.3 Analytical Engine

Charles Babbage, nineteenth century professor of Cambridge University, is considered the father of computers. Babbage designs a “Difference Engine” in the year 1822, which could produce reliable tables. In 1842, Babbage got new idea of “Analytical engine”, which was intended to be completely automatic. It was to be capable of performing the basic arithmetic functions for any mathematical problem, and it was to do so at an average of 60 additions per min.

2.1.4 Atanasoff- Barry computer

This electronic machine was developed by Dr. John Atanasoff to solve certain mathematical equations. It was called Atanasoff Barry computer (ABC). It is used 45 vacuum tubes for internal logic and capacitor for storage.

2.1.5 ENIAC

The electronic numerical integrator and calculator (ENIAC) was completed in1946. It was constructed at the Moore school of engineering of the University of Pennsylvania with funding from the U.S. Army. Weighting 30 tons and occupying a 30-by-50 foot space, the ENIAC was used to compute ballistic tables, predict the weather, and make atomic energy calculations.

2.1.6 EDSAC

The expansion of EDSAC is Electronic Delay Storage Automatic Calculator. In this machine, addition operation accomplished in 1500 micro seconds, and multiplication operation in 4000 micro seconds, the machine was developed by a group of scientists at Cambridge University mathematical laboratory.

2.1.7 UNIVAC

The expansion of UNIVAC is universal Automatic computer. The first business use of computer, in 1952, the International business Machines (IBM) corporation introduced the 701 commercial computers.

2.2 Modern computers

Modern computers are categorized according to their size and performance. Some of the modern computers are

- ✓ Micro Computer
- ✓ Mini Computer
- ✓ Mainframe Computer
- ✓ Super Computer

2.2.1 Micro Computer

Micro Computer is also called as personal computers. It is a small, single-user computer based on a micro processor. In addition to the microprocessor, a personal computer has a keyboard for entering data, a monitor for displaying information and storage device for saving data. These types of computers are used for small industrial control, process control, and where as storage and speed requirements are moderate.

2.2.2 Mini Computer

Mini computers are scaled up version of the micro computers with the moderate speed and storage capacity. These are designed to process smaller data words, typically 32 bit words. This type of computers is used for scientific calculations, research, data processing application and many others.

2.2.3 Mainframe computer

Mainframe computers are implemented using two or more central processing unit (CPU). These are designed to work at very high speeds with large data word length, typically 64 bit or greater. The data storage capacity of these computers is very high. This type of computers is used for complex scientific calculations, large data processing applications, and military defense control.

2.2.4 Super computer

These computers are basically multiprocessor computers used for the large scale numerical calculation required in application such as weather forecasting, robotic engineering, aircraft design and simulation.

2.3 Generation of computers

"Generation" in a computer talk is a step in technology it provides a frame work for growth of the computer industry. Originally, the term "generation" was used to distinguish between varying hardware technologies. Computers are evolved from 1942 to till date in five generations. They are

- ✓ First generation (1942-1955)
- ✓ Second generation (1955-1964)
- ✓ Third generation (1964-1975)
- ✓ Fourth generation (1975-1989)
- ✓ Fifth generation (1989- till date)333

2.3.1 First Generation (1942-1955)

The main function unit of first generation computers is "Vacuum Tubes" .It is used for developing the circuitry. It comprised of glass and filaments. These computers occupy very large space and consume large amount of electricity power. These computation times was calculated in milliseconds.

2.3.2 Second generation (1955-1964)

In this generation, vacuum tubes are replaced by transistor, which is a solid state device. Transistors are more reliable than vacuum tubes. It occupies less space compare to

vacuum tubes. These computation times was calculated in microseconds. These are less expensive and faster than the first generation computers.

2.3.3 Third generation (1964-1975)

Third generation computer were used Integrated Circuits(IC). IC is silicon chip that embeds an electronic circuit comprising of several components such as transistors, diodes and resistors etc., these computation times was calculated in nanoseconds. These are more

reliable and less expensive. In this generation, computers having high storage capacity and high processing speed.

2.3.4 Fourth generation (1975-1989)

This computer used Large Scale Integrated Circuits (LSI) and Very Large Scale Integrated Circuits (VLSI).These circuits are also termed as "Microprocessor chip". It integrates thousands of components on a single chip. These computers are very small in size. These are used for several applications like mathematical, modulating, computer aided designing etc.

2.3.5 Fifth generation (1989- till date)

The fifth generation computers are developed on the basis of a technique called Artificial Intelligence (AI). These are mainly used in robots. The input and output information for these computers will be in the form of speech and graphic images etc. These computers are used for some specialized applications.

3. Computer Hardware

The physical components of computer is called computer hardware, which includes mainly input devices, output devices , processor, Main memory(RAM) and secondary storage devices etc.

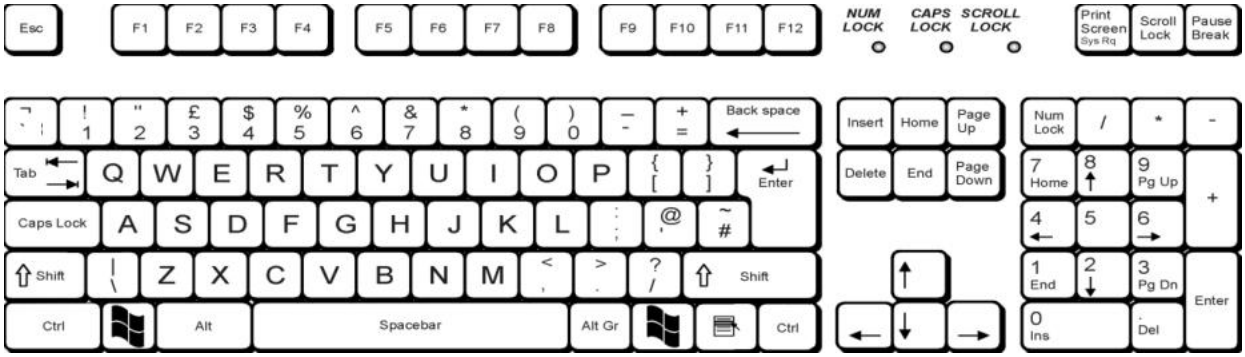
3.1 Input Devices

Input Devices are used to enter data and instructions into a computer. The most commonly used input devices are key board, mouse, scanner, joystick, and track ball etc.

3.1.1 Keyboard

Programs and data entered into the computer through a key board. When you press a letter or digit key on a key board, that character is sent to main memory and is also displayed

on the monitor at the position of the cursor. A standard keyboard includes alphanumeric keys, function keys, modifier keys, cursor movement keys, space bar and special purpose keys.



The *alphanumeric keys* include the alphabet keys and numeric keys. The *function keys* are the keys that help to perform a specific task such as searching a file or refreshing a web page. The *modifier keys* such as shift and control keys modify the case style of a character or symbol. The *cursor movement keys* include up, down, left, right keys and are used to modify the direction of the cursor on the screen. The *spacebar key* shifts the cursor to the right by one position. The special purpose keys such as page up, page down, home, end, insert, delete etc.

3.1.2 Mouse

A mouse is a handheld pointing device used to select an option. When you move the mouse around on your desktop a rubber ball attached to the mouse rotates and simultaneously moves the mouse cursor displayed on the monitor screen. The mouse is also known as a pointing device because it helps to change the position of the pointer or cursor on the screen. The mouse consists of two buttons, a wheel at the top and a ball at the button of the mouse. The wheel is used to scroll down in a document or web page.

3.1.3 Scanner

Scanner is a one kind of input device that converts documents and images as the digitized images understandable by the computer system. The images can be produced as black and white images, grey images or colored images. In case of colored images, an image is considered as a collection of dots with each dot representing a combination of red, green, blue colors varying in proportion. The Proportions of red, green, blue colors assigned to a dot are together called as color description.

3.1.4 Joystick

A joystick is also a pointing device. It is also used to move the cursor position on a monitor or screen. Its function is similar to the mouse and is used for playing games.

3.1.5 Track Ball

A track ball is also a pointing device and contains a ball, which can rotate in any direction. The user spins the ball in different directions to move the cursor on the monitor.

3.1.6 Space Ball

Space ball is an essential tool. Giving you the ability to manipulate 3D objects on the screen, while simultaneously controlling 3D camera angles & positions for viewing those objects.

3.2 Output Devices

The data, processed by the CPU, is made available to the end user by the output devices. The most commonly output devices are monitor, printer, speaker and plotter.

3.2.1 Monitor

A Monitor is the most commonly used output device. A monitor provides a temporary display of information that appears on it screen. It displays the text or picture in color or black and white depending on the type of the monitor. It is very similar to the television.

The monitors can be classified as,

- Cathode Ray tube (CRT)
- Liquid crystal Display (LCD)

The CRT monitors are large, occupy more space in the computer, where as LCD monitor are thin, light weighted and occupy lesser space. The monitor can be characterized by its monitor size and resolution. The monitor size is the length of the screen. The resolution of the monitor is also called the dot pitch.

3.2.2 Printer

Printers are the most popular output devices. Printer is an output device that produces a hard copy of information sent to it i.e., it prints the information or any result of the program on the paper.

The various types of printers are used in the market are generally categorized as

- Dot Matrix Printer
- Inkjet Printer
- Laser Printer

The *Dot Matrix Printers* are used in low Quality and high volume applications like invoice printing, cash register etc.

The *Inkjet printers* are slower than dot matrix printers and generate high quality photographic prints.

The *Laser printer* Quality is higher, and faster. A separate microprocessor and memory are built in to printer to interpret the data that it receives from the computer and to control the laser.

The printer driver software is used to convert a document to a form understandable by the computer. The performance of a printer is measured in terms of Dot per Inch (DPI) and Page per Minute (PPM) produced by the printer.

3.2.3 Speakers

Speakers are one of the most common output devices used with computer system. A speaker gives sound output from your computer. Some speakers are built into the computer and some are separate

The audio drivers need to be installed in the computer to produce the audio output. The sound card being used in the computer system decides the quality of audio that we listen using music CD's. The computer speakers vary widely in terms of quality and price.

3.2.4 Plotter

The plotter is another commonly used output device that is connected to a computer to print large documents, such as engineering or constructional drawings. Plotters use multiple ink pens or inkjets with color cartridges for printing.

3.3 Memory Devices

The memory devices are used to store programs and data. Usually, two types of memory devices are used to form a memory unit:

- ✓ Primary Memory (or) Main Memory
- ✓ Secondary Memory

3.3.1 Primary Memory (or) Main Memory

Main memory stores programs, data and results most computers have two types of main memory

- ✓ Random Access Memory
- ✓ Read only memory

3.3.1.1 Random Access Memory (RAM)

RAM is a temporary storage of program and data while they are being executed (carried out) by the computer. it is also temporarily stores such data as numbers, names, and even pictures while a program is manipulating them. RAM is usually volatile memory, which means that everything in RAM will be lost when the computer is switched off. in this memory, the data can be modified. in this memory, we can perform both read & write operations.

There are two types of RAMs

- Static RAM
- Dynamic RAM

Static RAM contains less memory cells per unit area. it less access time hence fast memories. Refreshing circuitry is not required. Cost is more.

Dynamic RAM contains more memory cells as compare to static RAM per unit area. It access time is grater then static RAM. Refreshing circuitry is required. Cost is less

3.3.1.2 Read Only Memory (ROM)

ROM is storage information permanently within the computer. The computer can retrieve (or read), but cannot store (or write), information in ROM, hence its name, read only because ROM is not volatile, the data stored there disappear when the computer is switched off. Start up instructions and other critical instructions are burned (write) into ROM chip at the factory. In this memory, the data cannot be modified.

Different types of ROMs

- ROM (or Mask ROM)
- PROM
- EPROM
- EEPROM

Mask ROM bits are stored permanently. This is done by manufacturers. This type of ROM can be programmed only one-by the manufacture. It had 1s and 0s actually burned into the integrated circuit. This technique was simple but inflexible, and it was often used to contain the startup code (bootstrap) for early microcomputers.

PROM is a Programmable Read Only Memory. The PROM is a memory array consisting of a grid of fuses. Typically, the blank PROM comes with all bits set to 1. During programming, the fuses that represent the zero bits are blown by the programming device, which sends high voltage pulses to destroy individual fuses. The PROM is a cheaper and more flexible approach than mask ROM, although each PROM can still be programmed only once. PROMs are reliable, permanent, and relatively fast.

EPROM is an Erasable Programmable Read Only Memory. The EPROM chip allows the stored data to erased and new data can be reprogrammed. The EPROM can be erased by exposure to strong ultraviolet light and programmed again. EPROM chips usually have a distinctive transparent quartz window on the top of the chip that exposes the transistors to the UV light.

EEPROM is an Electrically Erasable Programmable Read Only Memory. The contents of cells can be erased by the application of a high voltage. The EEPROM has largely supplanted all other types of ROM in the current generation of computing devices. The capacity of EEPROMs ranges up to hundreds of kilobits. This is now the preferred technology for storing the BIOS in personal computers.

3.3.2 Secondary Memory

Secondary storage Devices are used to store information permanently for future use. System typically store more information that will fit in memory. Secondary Storage devices are also called external storage devices. Some of the most commonly encountered secondary storage devices and storage media are:

- Magnetic tape
- Magnetic disk
- Optical disk or compact disk
- Floppy Disk

Magnetic tape is one of the most popular storage medium for large volumes of data. It is a sequential access device i.e., the data can be accessed sequentially (one by one) rather than randomly or directly. Like the recorder tape, the computer tape can also be erased and reused again and again.

Magnetic disk is a device used for mass storage of data. These are usually attached to their disk drives. The disk itself a thin platter of metal or plastic coated with a magnetic material. The data stored on a hard disk can be retrieved at a very fast speed, being a direct access device.

Optical Disk or Compact Disk is Most of today's computers are equipped with CD drives for reading data stored on compact disk. Some of these drives can also write data to CDs. A CD is coated with a material, which will change its reflecting property when high intensity laser beam is focused on it. CD-ROMs use long spiral tracks to store data serially. The track is divided into blocks (sectors) of the same size. One CD can hold 680MB of data. An increasingly common secondary storage device that uses similar technology is the Digital Video Disk (DVD), which can stores 4.7GB of data

Floppy disk is a movable storage disk used for storing data. You can make a copy of any important information from the hard disk in a floppy; this is known as keeping a backup of important information. A typical high-density floppy disk can store 1.44MB. You can use a floppy disk to store and move data easily from one PC to another PC.

3.4 Processor

The arithmetic and logic unit in conjunction with control unit is commonly called central processing unit (CPU).

The arithmetic and logical unit (ALU) is responsible for performing arithmetic operations such as addition, subtraction, multiplication, and division. Logical operations such as AND, OR,

NOT and etc. The data is transferred from the memory unit to the ALU, processed and returned to memory unit.

The control unit coordinates & controls the activities amongst the functional units. The basic function of control unit is to fetch the instructions stored in the main memory, identify the operations and the devices involved in it and accordingly generate control signals to execute the desired operations. it maintains order and directs the operations of the entire system.

4. Computer Software

A set of instructions is called a program. The set of programs with documentation is called software or the set of programs associated with a computer is called Software. There are two types of software's such as System software and Application software.

4.1 System Software

System software is the set of programs that coordinates the activities and functions of the hardware and other programs throughout the computer system. Each type of system software is designed for specific CPU. Types of system software include operating system, some utility programs.

4.1.1 Operating System

An operating system is a setof programs that controls the computer hardware, manages of allocation of resources and acts as an interface between user and computer. Operating system can control one or more computers, or they can allow multiple users to interact with one computer.

The operating system plays a central role in the functioning of the complete computer system. It is usually stored on disk, after you start or boot-up a computer system, some portions of the operating system are transferred to main memory as they are needed.

The following shows the list of some of the operating system's responsibilities:

1. Communicating with the computer user: receiving commands and carrying them out or rejecting them with an error message.
2. Managing allocation of memory, allocation of processor time, and of other resources for various tasks.

3. Collecting input from the keyboard, mouse and other input devices, and providing this data to the currently running program.
4. Conveying program output to the screen, printer, or other output device.
5. Accessing data from Secondary storage devices and writing data to secondary storage devices.

In addition to these responsibilities, the operating system of a computer with multiple users must verify each individual's right to use the computer and must ensure that each user can access only data for which he or she has proper authorization.

Some of the widely used operating systems are MS-DOS,UNIX, WINDOWS -95/98/XP, WINDOWS-NT, WINDOWS 7, LINUX etc.,. An OS that uses a command-line interface displays a brief message, called a PROMPT. That indicates its readiness to receive input and the user then types a command at the keyboard.

In contrast, operating system with a Graphical User Interface (GUI) provides the user with a system of icons and menus. To issue commands, the user moves the mouse, track ball to point to the appropriate icon or menu selection and pushes a button once or twice.

4.2 Application Software

Application programs are developed to assist a computer user in accomplishing specific tasks. For example , a word processing application that help to create document, a spread sheet application that help to automate tedious numerical calculations and to generate charts that depict data and a data base management application that assist in data storage and quick keyword-based access to large collections of records etc. Application software is used mainly for commercial purpose.

5. Algorithm

An algorithm is defined as *a finite set of steps that provide a chain of action for solving a definite nature of a problem* or *a method of representing the step by step procedure for solving a problem.*

An algorithm is a well organized, prearranged, and defined textual computation module that receives some values or set of values as input and provides a single value or set of values as

output. Each step in the algorithm should be well defined this will enable the reader to translate each step into a program.

These well defined computation steps are arranged in sequence that processes the given input into output. An algorithm is said to be accurate and truthful only when it provides the exact required output. Lengthy procedure is subdivided into small parts as this step makes it easy to solve a given problem. Every step is known as an instruction.

For Examples;

I. To establish a telephone communication between two subscribers, the following steps are to be followed:

```
Start
   1. Dialed the phone number.
   2. Phone ring at the called party.
   3. Caller waits for response.
   4. Called party pick up the phone.
   5. Conversation begins between them.
   6. Release of connection.
Stop
```

II. To make a cup of tea, the following steps are to be followed:

```
Start
   1. Boil water.
   2. Put tea powder in the bowl.
   3. Pour boiled water in the bowl.
   4. Wait for three minutes.
   5. Boil milk.
   6. Put boiled milk in a cup.
   7. Add sugar to the cup.
   8. Empty the bowl in the cup.
   9. Stir the cup with a spoon.
Stop
```

III. Write an Algorithm for finding Simple Interest.

```
Start
   1. [Read input values]
        Read p, t, r
```

```
    2. [Calculate Simple Interest]
         Compute Si = (p * t * r) / 100
    3. [Display the Result]
         Print Si
Stop
```

5.1 Characteristics of an algorithm

- ✓ Finiteness
- ✓ Definiteness
- ✓ Effectiveness
- ✓ Generality
- ✓ Input/output

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

5.1.2 Definiteness: Each step of the algorithm must be precisely defined i.e., each step must be clear and unambiguous.

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

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

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

5.2 Efficiency of an Algorithm

Efficiency of an algorithm means how fast it can produce the correct result for the given problem. The efficiency of an algorithm depends upon its time complexity and space complexity. The complexity of an algorithm is a function that provides the running time and space for data, depending on size provided by user.

The two import factors for judging the complexity of an algorithm are as follows:

- ✓ Time Complexity
- ✓ Space Complexity

5.2.1. Time Complexity: Time complexity of an algorithm refers to the amount of time required by an algorithm for its execution. This time includes both compile time and run time.

5.2.2. Space Complexity: Space complexity of an algorithm refers to the amount of memory space required by the algorithm for its execution and generation of the final output.

5.3 Some of the words used while writing the algorithm are

- ✓ Start/Begin
- ✓ Read/Accept
- ✓ Print/Write
- ✓ Compute/Calculate
- ✓ Stop/End

5.3.1 Start/Begin: The start/Begin is the words used at the beginning of the algorithm.

5.3.2 Read/Accept: These words are used to input the data from keyboard or to data from backup storage devices.

5.3.3 Print/Write: These words are used to output the information onto the screen or to write the information onto the backup storage devises.

5.3.4 Compute/Calculate: These words are used at expression evaluation or mathematical calculations.

5.3.5 Stop/End: These words are used at the end of the algorithm.

5.4 Analyzing an algorithm

Analyzing an algorithm refers to calculating or guessing the resources such as computer memory, processing time, logic gate and so on. The analysis can also made by reading the algorithm for logical accuracy, tracing the algorithm, implementing it, and checking with some data and with mathematical technique to conform its accuracy.

It is very essential to consider the factors such as time and space requirements of an algorithm. An efficient algorithm must be developed utilizing minimum system resources such as CPU time and memory.

5.5 Rate of growth

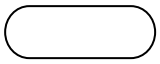
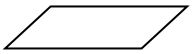
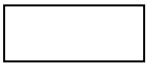
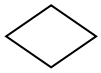
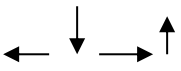
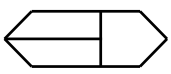
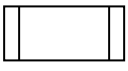
In practice, it is not possible to predict the execution time of an algorithm based on a simple analysis of the algorithm. The execution time depends upon the machine and the way of implementation.

6. Flowchart

Flowchart can be defined as *a diagrammatic or visual or Graphical representation of an algorithm*. A flowchart is an alternative technique for solving a problem. A completed flowchart enables to organize a problem into plan of actions.

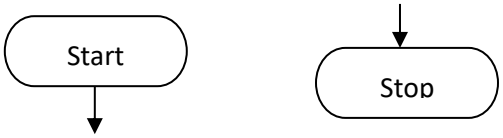
It is a working map of a final product. The flowchart is an easy way to solve the complex designing problem, to understand and analyze the problem. A flowchart is a set of symbols that indicate various operations in the program. A pictorial representation of textual algorithm is done using a flowchart.

Commonly used symbols in flowchart

S. No	Geometrical Figure	Name	Function
1.		Oval	Start and Stop
2.		Parallelogram	Input or Output
3.		Rectangle	Processing
4.		Diamond	Decision making
5.		Arrows	Connections
6.		Circle	Continuation
7.		Hexagon	Repetition/ looping
8.		----------	Pre defined process

6.1 Start and Stop

The start and stop symbol indicate both the beginning and the end of the flowchart. This symbol looks like a flat oval as shown in the figure.



Only one flow line is combined with this kind of symbol. We write start, stop or end in the symbols of this kind. Usually this symbol is used twice in a flowchart, i.e., at the beginning and end.

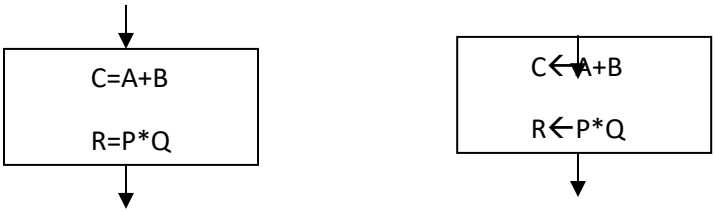
6.2 Input or Output

The input/ output symbol looks like a parallelogram as shown in figure below. The input/ output symbol is used to input and output the data.

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

6.3 Process symbol

The symbol of process block should be shown by a rectangle as shown in the figure below. It is usually used for data handling and values are assigned to the variables in this symbol.



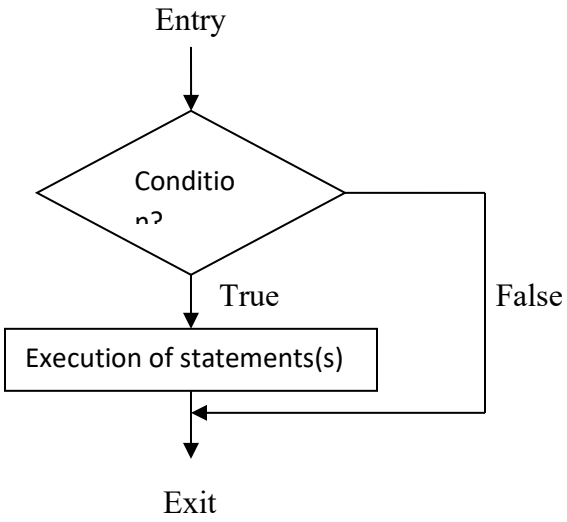
The operations mentioned within the rectangular block will be executed when this kind of blocks is entered in the flowchart. Sometimes arrow can be used to assign the value of a variable to another. The value indicated at its head is replaced by the tail value. There are two flow lines connected with the process symbol. One line comes to the symbol and another line goes from this symbol.

6.4 Decision or Test Symbol

The decision symbol is diamond shaped. This symbol is used to take one of the decisions. Depending on the condition the decision block selects one of the alternatives. While solving a problem, one can take a single alternative or two or multiple alternatives, depending upon the situation.

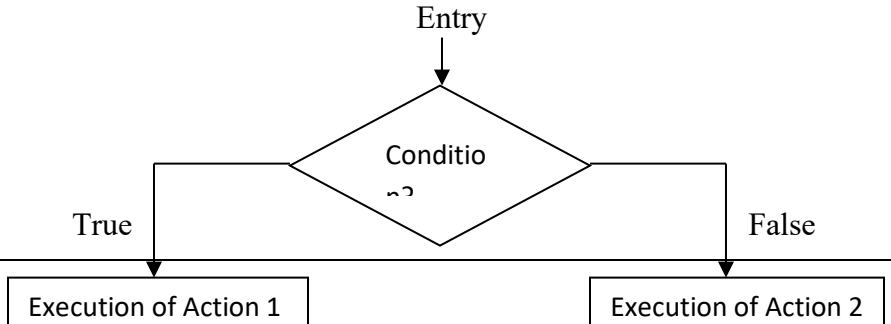
6.4.1 Single Alternative Decision

Here more than one flow lines can be used depending upon the condition. It is usually in the form of a 'yes' or 'no' question, with branching flow lines depending upon the answer. With a single alternative, the flow diagram will be as per below.



6.4.2 Two Alternative Decisions

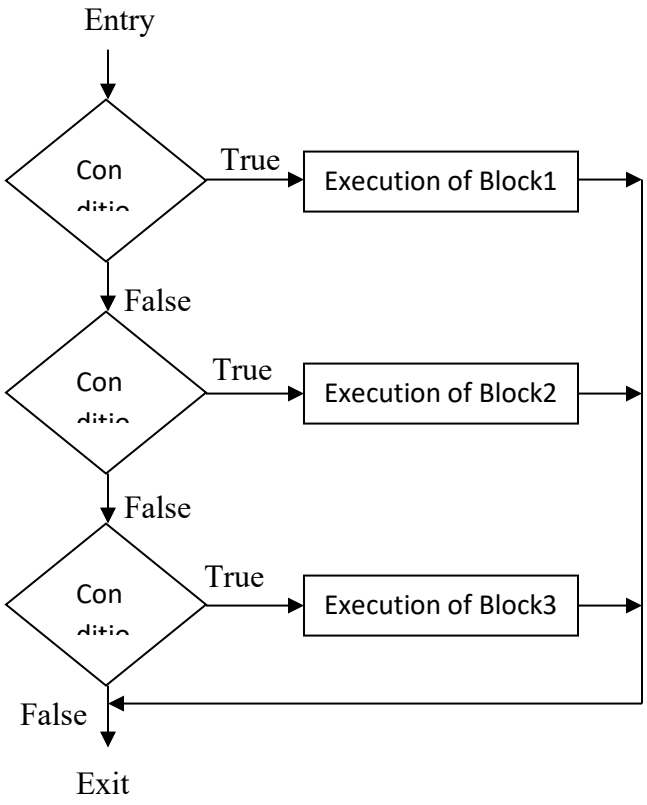
The below figure two alternative paths have been shown. On satisfying the condition statement(s) pertaining to action1 will be executed, otherwise the other statement(s) for action2 will be executed.



Exit

6.4.3 Multiple Alternative Decisions

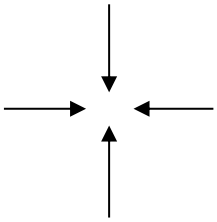
The below figure, multiple decision blocks are shown. Every decision block has two branches. In case the condition is satisfied.



Execution of statement of appropriate blocks take place, otherwise next connection will be verified. If condition 1 is satisfied then block1 statements are executed. In the same way, other decision blocks are executed.

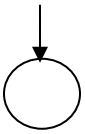
6.5 Connections

The shown in the below figures are connections. The connector is also called the flow line. Arrows (connections) indicating the sequence of steps or flow of control. The connections are used to connect one symbol to another symbol. For example; this is used to connect the start symbol to process symbol and any more symbols. Connection is used to connect any direction like up, down, left and right.

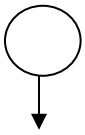


6.6 Connector symbol

The connector symbol has to be shown in the form of a circle. It is used to establish the connection, whenever it is impossible to directly join two parts in a flow chart. A connector can be used for joining the two parts. Only one flow line is shown with this symbol. Only connector names are written inside the symbol, that is, alphabets or numbers.



Connector for connecting to the next block

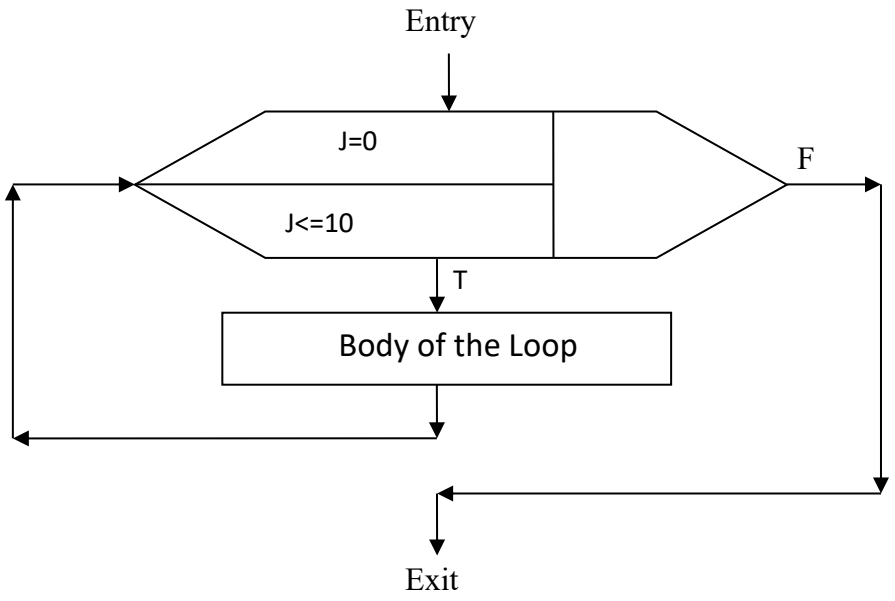


connector that comes from the previous block

6.7 Loop symbol

This symbol looks like a hexagon. This symbol is used to implementation of loop only. Four flow lines are associated this symbol. Two lines are used to indicate the sequence of the program and remaining two are used to show the looping area, that is, from the beginning to the end.

The shown in the below figure illustrate the working of the *for* loop, the variable 'J' is initialized to '0' and it is to be incremented by a step of 1 until it reaches a final value 10. For every increased value of 'J', body of the loop is executed. This process will be continued until the value of J reaches 10.



6.8 Pre defined process symbol

It is used for subroutine or pre defined process and it defines the function definition. Only one flow line is combined with this kind of symbol.

One flow line is used to the symbol and flow line goes from this symbol. Sub-routine (function) used to indicate a process, which is defined elsewhere.

7. Software development method

Programming is the problem solving activity. For this, the programmers use the software development method.

The phases or steps that are that are involved in this method are:

- ✓ Specify the problem requirements.
- ✓ Analyze the problem.
- ✓ Design the algorithm to solve the problem.
- ✓ Implement the algorithm.
- ✓ Test and verify the completed program.

- ✓ Maintain and update the program.

7.1 Problem

Specifying the problem requirements forces us to state the problem clearly and unambiguous and to gain clear understanding of what is required for its solution. Our objective is to eliminate unimportant aspects and zeros in on the root problem.

7.2 Analysis

Analyze the problem involves indentifying the problem,

a) Input - The data you have to work with
b) Any additional requirement or constraints on the solution.
c) Output - The desired results.

At this stage, you should also determine the required format in which the results should be displayed and develop a list of problem variables and their relationships. These relationships may be expressed as formulas. All the identified requirements are documented so as to avoid any doubt or uncertainties pertaining to the functionality of the program.

7.3 Design

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

First, write the algorithm in *top-down design* (also called divide and conquer). In this, first list the major steps, or sub problems, that need to solve. Then solve original problem by solving each of its sub problems.

Most of the algorithm will be like,

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

Once the sub problems are known, each one can be solved individually. This can be done through a process called *algorithm refinement*.

7.4 Implementation

Implementing the algorithm involves writing it as a program. This step involves writing the instructions or code for the program on the basis of the design document created in the previous design step i.e., converting each step of algorithm into one or more statements in a programming language. The choice of the programming language in which the program will be developed is made on the basis of the type of program.

7.5 Testing

Testing and verifying the program that requires testing the completed program to verify that, it works as desired. Run the program several times using different sets of input data to make sure that it works correctly for every situation provided in the algorithm.

7.6 Maintenance

Maintaining and updating the program involves modifying a program to remove previously undetected errors and to keep it up-to-date as government regulations or company policies change. Many organizations maintain a program for five years or more often after the programmers who originally coded it have left or moved on to other positions. A disciplined approach is essential if you want to create programs that are easy to read, understand, and maintain.

8. Applications of the software development method

It is begin with a problem statement. As part of the problem analysis, we identify the data requirements for the problem, indicating the problem input and the desired outputs. Next we design the initial algorithm and then implement the algorithm as a C program. Finally test and verify the program for different sets input data.

*8.1 **Case Study I**: finding the Simple Interest*

8.1.1 Problem definition

Finding the Simple Interest with the given information i.e. Principle amount, Rate of Interest, and Time.

8.1.2 Analysis

In this step, it requires to identify the input (i.e., Principle amount, rate of interest and time) and desired output (i.e., Simple interest). The requirements and relevant formulas to solve a problem are listed below:

- Problem Input / Data requirements: p, t, r (Principle amount, rate of interest and time)
- Relevant formula to solve a problem: Si = (p * t * r)/ 100
- Problem desired output: Si (Simple Interest)

8.1.3 Design

Next formulate the algorithm that solves the problem.

Algorithm

1. Get the principle amount(p), rate of interest(r), time(t)
2. Calculate the Simple interest(si)
3. Display the Simple interest(si)

Now decide whether any steps of the algorithm need further refinement or whether they are perfectly clear as stated. Here, Step 2 requires the refinement.

Step 2 Refinement

2.1. The simple interest is the combination of multiplied value of principle amount (p), rate of interest (r) and time (t) by 100.

So, the refinement algorithm will be,

Algorithm with refinement

1. Get the principle amount(p), rate of interest(r), time(t)
2. Calculate the Simple interest(si)
   2.1. The simple interest is the combination of multiplied value of principle amount (p), rate of interest (r) and time (t) by 100.
3. Display the Simple interest(si)

8.1.4 Implementation

To implement the solution, you must write the algorithm as a C program. The C program along with a sample execution is shown below:

```
/*Program to Calculate the Simple interest */
        #include<stdio.h>
        #include<conio.h>
        void main()
```

```
{
 float p, r ,t, si;
clrscr();
printf("\n Enter Input Values:");
scanf("%f%f%f",&p,&r,&t);
si = (p*r*t)/100;
printf("\n The Simple Interest is %f",si);
}
```

8.1.5 Testing

To verify that the program works properly, enter more test values of principle amount, rate of interest and time as shown below

Case 1:

Enter Input values: 1000 10 2

The Simple Interest is: 200.0000

Case 2:

Enter Input values: 2000 5 4

The Simple Interest is: 400.0000

*8.2 **Case Study II**: converting miles to kilometers*

8.2.1 Problem definition

Find that given distances in kilometers and some that use miles. Write a program that performs the necessary conversion.

8.2.2 Analysis

The First step in solving this problem is to convert distance measurements in miles to kilometers. Therefore, the problem input is *distance in miles* and the problem output is *distance in kilometers*.

So, the data requirements and relevant formulas are as,

- Problem Input / Data requirements: miles
- Relevant formula to solve a problem: 1 mile=1.609 kilometers
- Problem desired output: kilometers

8.2.3 Design

Next, formulate the algorithm that solves the problem.

Algorithm

1. Get the distance in miles.
2. Convert the distance to kilometers.
3. Display the distance in kilometers.

Now decide whether any steps of the algorithm need further refinement or whether they are perfectly clear as stated. Here, Step 2 requires the refinement.

Step 2 Refinement

2.1. The distance in kilometers is 1.609 times the distance in miles.

So, the refinement algorithm will be,

Algorithm with refinement

1. Get the distance in miles (m).
2. Convert the distance to kilometers (km).
   2.1. The distance in kilometers is 1.609 times the distance in miles.
3. Display the distance in kilometers (km).

8.1.4 Implementation

To implement the solution, you must write the algorithm as a C program. The C program along with a sample execution is shown below:

```
/*Program to Calculate the Simple interest */
                #include<stdio.h>
                #include<conio.h>
                void main()
                {
                 float m, km;
                clrscr();
                printf("\n Enter Input Values:");
                scanf("%f",&m);
                km = 1.609*m;
```

```
    printf("\n The kilometers are %f",km);
    }
```

8.1.5 Testing

To verify that the program works properly, enter more test values of principle amount, rate of interest and time as shown below

Case 1:

Enter Input values: 10

The kilometers are 16.0900

Case 2:

Enter Input values: 20

The Simple Interest is: 32.1800

9. Computer Languages or Programming Languages

The operations of a computer are controlled by a set of instructions (called a computer program). These instruction are written to tell the computer that what operations to perform, where to locate data, how to present results and when to make certain decisions.

The language used in the communication of computer instructions is known as the programming language. The computer has its own language and any communicating with the computer, must be in its language or translated into this language.

Three levels of programming languages are available. They are:

1. Machine Language
2. Assembly (or Symbolic)Language (Low Level Languages)
3. High Level Languages

9.1 Machine Language

As computers made of two-state electronic devices, they can understand only pulse and no-pulse (or '1' and '0') conditions. Therefore all instructions and data should be written using binary codes i.e., 1 and 0. This binary coded program is called machine language.

It poses two problems for the user: First, it is difficult to understand and remember the various combinations of 1's and 0's are representing numerous data and instructions. Secondly, since every machine has its own machine language, the user cannot communicate with other

computers (if he does not know its language).Machine Languages are usually referred to as the first generation languages.

Draw backs

1. It is not standardization.
2. There is a different machine language for every type of CPU
3. It is not easy for programming

9.2 Assembly Language

The Assembly language introduced to reduce programming complexity and provided some standardization to build an application. The assembly language is also referred as the second-generation programming language, is also a low-level language. In an assembly language, the 0's and 1's of machine language are replaced with abbreviations or mnemonic code. The assembly language program is converted into the machine code with the help of an assembler. *The main advantages of an assembly language over a machine language are:*

1. As we can locate and identify syntax errors in assembly language, it is easy to debug it.
2. It is easier to develop a computer application using assembly language in comparison to machine language.
3. Assembly language operates very efficiently

Draw backs

1. it is also different for every type of CPU
2. It is easier than machine language but till it is not user compatible.

9.3 High Level Language:

The High level language is a machine independent programming language that combines algebraic expressions and English symbols. Instead of dealing with registers, memory addresses and call stacks, a programmer can concentrates more on the logic to solve the problem with help of variables, arrays or Boolean expressions. High level languages like COBOL, Pascal, FORTRAN, and C are more abstract, easier to use, and more portable across platforms. There are many High level languages available. Some of the high level languages are listed in the following table:

Language	**Application Area**	**Origin of Name**
FORTRAN	Scientific Programming	**For**mula **Tran**slation
COBOL	Business Data Processing	**Co**mmon **B**usiness **O**riented **L**anguage
C	Systems programming	Predecessor language was named B

C++	Supports Objects and Object Oriented Programming	Incremental modifications of C
Java	Supports Web Programming	Originally named as "Oak"
Prolog	Artificial Intelligence	**Log**ic **Pro**gramming
LISP	Artificial Intelligence	**Lis**t **P**rocessing

10. Introduction to C Language

10.1 history of C language

C language was developed by **Dennis M Ritchie**, in 1972, at AT &T Bell labs. C language was derived from B language which was developed by **ken Thomson** in 1970. B language was adopted from the language **BCPL (basic combined programming language)** which was developed by martin Richards at Cambridge University. The language **B** named as so by borrowing the first initial from BCPL language. BCPL was a response to difficulties with its predecessor Combined Programming Language (CPL), created during the early 1960s. Richards created BCPL by "removing those features of the full language which make compilation difficult". CPL was developed jointly between the Mathematical Laboratory at the University of Cambridge.

It was heavily influenced by **ALGOL 60**, but instead of being extremely small, elegant and simple, CPL was intended for a wider application area than scientific calculations and was therefore much more complex than and not as elegant as ALGOL 60. C is a general-purpose programming language initially developed by Dennis Ritchie between 1969 and 1973 at AT&T Bell Labs. Its design provides constructs that map efficiently to typical machine instructions,

and therefore it found lasting use in applications that had formerly been coded in assembly language, most notably system software like the **Unix** computer **operating system**

Year	Name of the Language	Developed by
1960	ALGOL	International Committee
1963	CPL	Cambridge University
1967	BCPL	Martin Richards at Cambridge University
1970	B	Ken Thomson at A T & T Bell Labs
1972	C	Dennis M Richie at A T &T Bell Labs

11. Structure or General form of c program

C is a procedural oriented programming language. A c programming consists of various sections as shown in the following figure.

Document section
Preprocessor directive section
Global declaration section

main( ) Section
{
Declaration part
Executable part
}
 	
User defined function ( ) Section
{
Statement(s)
}

11.1 Document section

The document section consists of a set of comment lines giving the name of the program, the author and other details which the programmer would like to use later. In c, comments begin with the sequence /* and terminated by */ anything that is between the beginning and ending comments symbols is ignored by the compiler.

For example

/* sample program */

11.2 Preprocessor Directive Section

The statements begin with # symbol, and are called the preprocessor directives. These statements direct the c pre processor to include the *header files* and *symbolic constants* in to a c program.

In *Header files*, a c program depends upon some header files for functions that are used in the program. Each header file by default has an extension '.h'. The file should included using #include. Path name must either be enclosed by *double quote marks* and *angle brackets*.

For example

1. #include<stdio.h>

In the above example, the < > tells the preprocessor to search for the include file in a include directory or directories.

2. #include "stdio.h"

In the above example, the double quote marks " " indicate that the current directory should be checked for the header file first. If it is not found, the special directory or directories should be checked.

In *symbolic constants*, the definition section defines all the symbolic constants by using #define directives. ANSI C allows you to declare constants. The #define directive is used to tell the preprocessor to perform a search and replace operation.

For Example

1. #define PI 3.143
2. #define KM_PER_MILES 1.609

In the above examples, the preprocessor will search through the source file and replace every instance of token PI with 3.143 and KM_PER_MILES with 1.609.

11.3 Global declaration section

This section declares some variables that are used in more than one function. These variables are known as global variables. These global variables should be declare in global declaration section, i.e., outside of the all the functions.

11.4 main( ) section

Every program written in c language must contain the main( ) function. The main() is starting point of every 'C' program. Execution of the program always begins with the function main( ). The program execution starts from the opening brace ({) and ends with closing brace (}) between these two braces, the programmer should declare declaration & execution part. The function main( ) should be written in lower case letters.

11.4.1 Declaration part

The declaration part declares all variables that are used in the executable part. Initialization of variables is also done in this section. Initialization means providing initial values to the variables.

For examples are,

```
int a,b; this is for declaration
int a=2,b=3;  this is for initialization
```

11.4.2 Executable part

This part contains the statements following declaration of the variables. This part contains a set of statements or a single statement. These statements are enclosed between braces. They may be input output statements, arithmetic statements, control statement and other statements.

For examples are,

```
c=a+b;
sum=sum+i;
```

11.5 user defined function

The function defined by the user is called user defined functions. These functions are generally defined after the main( ) function. They can also be defined before the main( ) function.

12. Programming Rules

1. All statement should be written in lower case letter. Upper case letters are only used for symbolic constants.

2. Blank spaces may be inserted between the words. This improves the reliability of the statement. However, it is not used while declaring a variable, keyword, constant and functions.
3. It is not necessary to fix the position of statement in the program, that is, the programmer can write the statement anywhere between the two braces following the declaration part. The user can also write two or more statement in one line separating them with a semicolon (;). Hence, it is often called a free from language.
   a=b+c;
   d=b*c;
4. The opening and closing braces should be balanced.

13. C character set

A character denotes any alphabet, digit or special should used to represent information. The valid character set of c is shown below

Alphabets: uppercase letters (A to Z) Lowercase letters (a to z)	*Special symbols:* , comma . period : colon ; semicolon + plus sign - minus sign ‘ apostrophe “double quote marks | pipeline ! exclamation mark ? question mark @ at the rate # hash % percent ^ caret & ampersand * asterisk ~ tilde (left parenthesis) right parenthesis {left brace }right brace [left bracket] right bracket / slash \ back slash = equal sign _ underscore < less than > greater than
Digits: 0 through 9	
White Space: Blank space Horizontal tab Vertical tab New line Form feed	

14. C Tokens or C language elements

The basic and the smallest individual units of a C program are called C tokens. There are categorized into following categories.

- ✓ Keywords
- ✓ Identifiers
- ✓ Constants and variables
- ✓ Operators

14.1 Keywords

The c keywords are reserved words by the compiler. All keywords have been assigned fixed meaning. The keywords cannot be used as variable names because they have been assigned fixed jobs. There are 32 keywords available in C. all the keywords appear in lower case.

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

14.2 Identifier

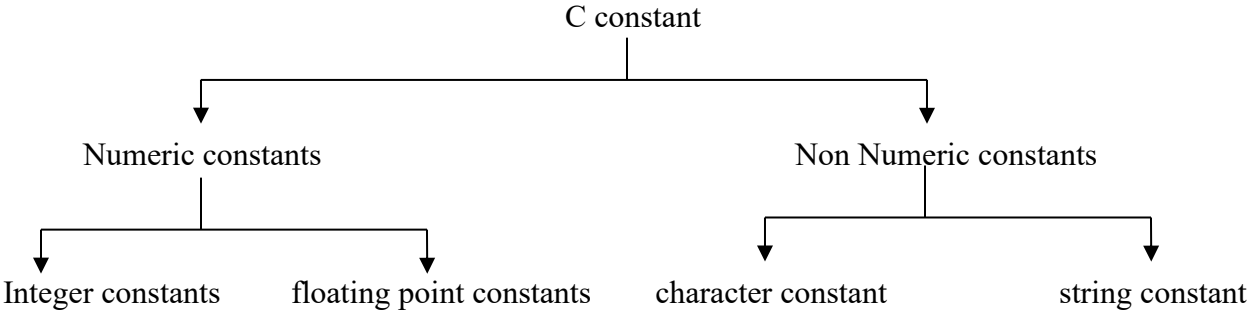
Identifiers are names of variable, functions and arrays. They are user defined names consisting of sequence of letters and digits.

Rules for forming identifier names

1. Identifier names must be a sequence of letters, digits and underscore (_).
2. Identifier must begin with a letter.
3. Uppercase and lowercase identifiers are different in C.
4. No special characters or punctuation symbols are allowed except the underscore (_).
5. No two successive underscores are allowed.
6. Keywords should not be used as identifiers.

14.3 constants

The Quantity which does not change during the execution of a program is known as constant. There are two types of constants.



14.3.1 Numeric constant

The numeric constants are two types.

1. Integer constant
2. Floating point constant

14.3.1.1 Integer constant

An integer constant refers to a sequence of digits. There are three type of integer constants namely Decimal integer (base 10), Octal integer (base 8), and Hexadecimal integer (base 16). *Decimal integers* consist of a set of digits, 0 through 9, preceded by an optional – or + sign. Valid examples of decimal integer constants are

```
245,    +375,    -965,  0
```

An *Octal integer* constant consists of any combination of digits from the set 0 through7, with a leading 0. Valid examples of octal integer constants are

```
037,    0435,    0
```

Hexadecimal integer constant consists of a sequence of digits preceded by 0x or 0X. They may also include alphabets A through F. The letter A through F represents the numbers 10 through 15. Valid examples of hexadecimal integer constants are

```
0x89,  0X7F,  0x,  0xAB
```

We rarely use octal and hexadecimal numbers in programming.

14.3.1.2 Floating point constants

Real (or Floating point) constants have a decimal point or an exponential part or both. Real constants can be represented in either Decimal notation or Exponential (Scientific) notation.

Decimal Notation in Decimal notation, the number is represented as a whole number followed by a decimal point and a fractional part. It is possible to omit digits before or after the decimal point. The following are examples of valid real or floating point constants

```
.986,   235.45,    -.75,    +.6
```

Exponential (or Scientific) Notation is useful in representing numbers whose magnitudes are very large or very small. The exponential notation consists of a mantissa and an exponent.

The number 7500000000 can be written as 75e9 or 0.75e11.
Similarly 0.00000000045 can be written as 0.45e-9
The following examples are valid constants:

3.4e4, 2.8E-3, -0.35e2

The rules governing exponential representation of real constants are given below:

- The mantissa is either a real number expressed in decimal notation or an integer
- The mantissa can be preceded by a sign
- The exponent is an integer preceded by an optional sign
- The letter 'e' can written in lowercase or upper case
- Embedded whitespace is not allowed

By default, real constants are assumed to be double.

14.3.2 Non Numeric constants

Non numeric constants are two types

1. Single character constant
2. String constant

14.3.2.1 Single Character Constants:

A single character constant (or simply character constant) contains a single character enclosed with in a pair of single quote marks. Examples of character constants are

'a', 'S', '4', '$'

14.3.2.2 String Constants

A String constant is a group of characters enclosed in double quotes. Examples of String constants are

"abc", "program", "b", "WELCOME"

14.3.3 Backslash character constant or escape sequence

C supports some special backslash character constants that are used in output functions. Backslash constant is a combination of two characters in which the first character is always the backslash (\) and second character can be any of the character a, b, f, n, r, t, v, ', '' ,\ , 0. The backslash characters constant are also called the escape sequences. The backslash constants are used in the output statements.

Constant	Meaning
\a	Beep sound(Alert)
\b	Back space
\n	New line
\t	Horizontal tab
\r	Carriage return
\v	Vertical tab
\f	Form feed

\′	Single quote
\″	Double quote
\\	Backslash
\0	Null

Note that each one of them represents one character, although they consist of two characters, these characters combinations are known as escape sequences.

14.4 variables

The quantity that changes during the execution of the program is called a variable. The variables are the names given to identify the specific program elements. Therefore variables are also called identifiers. The variables represent a particular memory location where data can be stored. They used to denote constant, functions, arrays, fields of structure, name of the file.

Examples: sum, area, length, name, age, city, etc.

Rules for forming variable names

1. Identifier names must be a sequence of letters, digits and underscore (_).
2. Identifier must begin with a letter.
3. Uppercase and lowercase identifiers are different in C.
4. No special characters or punctuation symbols are allowed except the underscore (_).
5. No two successive underscores are allowed.
6. Keywords should not be used as variables.
7. Writing the variable name in small letters is good programming practice.
8. There is no limit on the number of characters in a variable name.

Examples for valid variable name
Marks, TOTAL_MARKS, grocess_sal_2006, area_of_circle(), num[20]

Examples of invalid variables names are

Cube's_ volume	Illegal character(')
TOTAL MARKS	Blank space are not allowed
8ab	First digit is not allowed
Grocess-salary-2004	Special characters are not allowed
grocess__ salary__2004	Two successive underscore (_) not allowed.

14.4.1Variable declaration

All variables must be declared before they are used in a c program. The declaration does two things.

1. It tells the compiler what the variable name is.
2. It specifies what type of data the variable will hold.

The declaration is made in the declaration part of a C program. The syntax for declaring a variable is as follows.

data type	variable list	semicolon (;)

Where,

Data type → is a basic data type such as int, char, float or double.

Variable list→ one or more variables of data type. These variables must be separated by

Commas.

Semicolon (;) → a delimiter of this declaration.

For example

int length;	int a,b;
float area;	float p,q;
char ch;	char ch,c;

14.4.2 Assigning a values to variables or initialization of variable

We know that variable represent some memory location, where the data is stored each variable is associated with one or more values. The process of giving values to variable is called the assignment of values. The assignment operator '=' is used to assign a value to a variable. Its syntax is as follows:

Data type	Variable_name	=	Value	Semicolon (;)

or

Variable_name	=	Value	Semicolon (;)

Where,

Data type → is a basic data type such as int, char, float or double.

Variable_name → represent the memory location where the 'value' should be stored.

= → an assignment operator.

Value → is content or a variable.

Semicolon (;) → a delimiter of this declaration.

There are two methods of assigning values to variables. In first method, the initial values can be assignment to variables within the declaration. In second method, the initial values are assigned to the variables in the executable part of a program. Assigning variables with the declaration is called initialization. In this case, the declaration must consist of a data type followed by a variable name, an equal sign & a number appropriate to the data type & finally a semicolon.

Example of initialization

```
int x=1;            float sum=30.0;
char ch='y';        double r=0.123e-3;
```

Assignment with in an executable part does not include data type.

Examples:

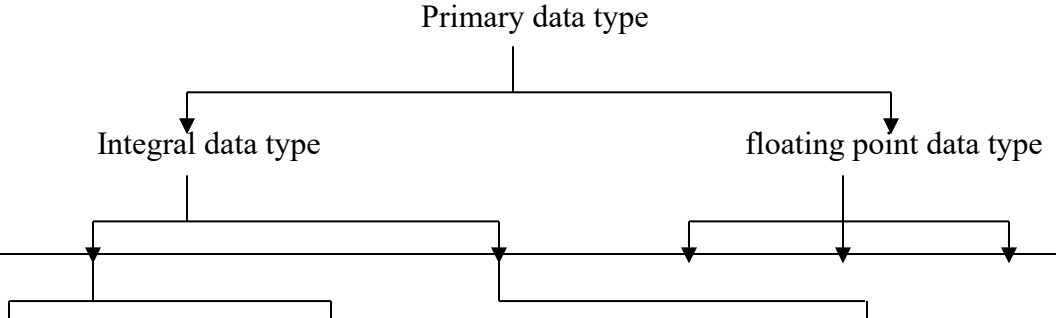
X=10; sum=300.00; name="murahari reddy"; ch='y';

15. Data type

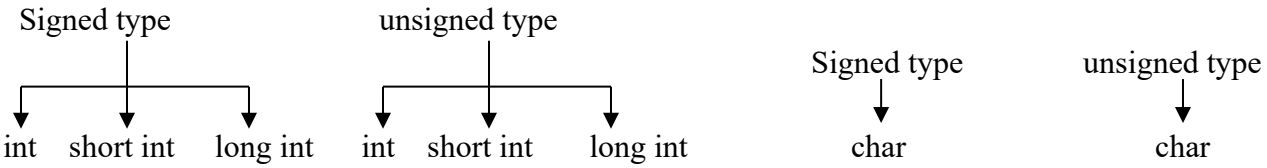
Data types indicate the type of data that a variable can hold. The data may be numeric or non numeric in nature. C language is rich in its data types. C compilers support a variety of data types. This enables the programmer to select the appropriate data type as per the need of the application. In C, the data types are categories into:

- ✓ Primary Data types or Built-in Data types.
- ✓ Derived Data type.
- ✓ User defined Data type.
- ✓ Empty Data set.

15.1. Primary Data type



Integer types Character types float double long double



All C compilers support fundamental data types, namely integer (**int**), character (**char**), floating point (**float**), and double precision floating point (**double**). Many of them also offer extended data types such as **long int** and **long double**.

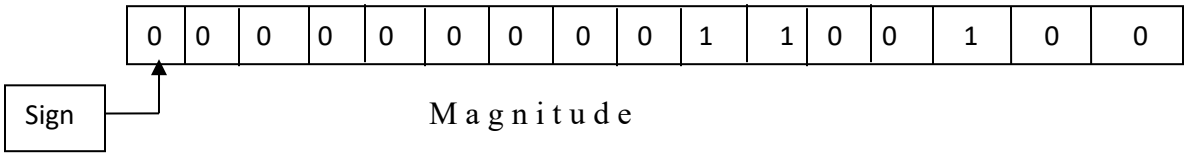
15.1.1 Integer types

Any integer number is a sequence of digits without a decimal point. Generally, integers occupy one word of storage, and since the word sizes of machines vary (typically, 16 or 32 bits) the size of an integer that can be stored depends on the computer. Integer data type can be divided into two types such as **Signed integer** and **unsigned integer**.

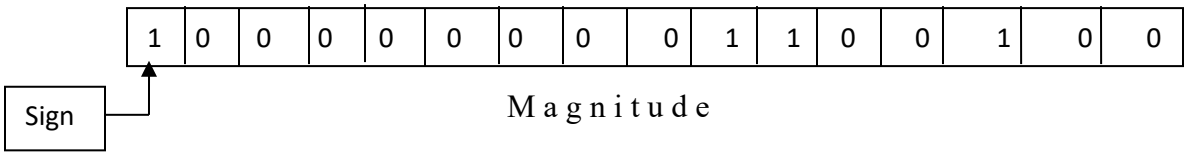
Signed integer

The signed integers are use 15 bits for storing the magnitude of a number and 1 bit is used for sign ie. Left most bit (16^{th} bit). The number negative when the left most bit (16th bit) is 1, otherwise the number is positive. If we use a 16 bit word length, the size of the integer value is limited to the range -32768 to 32767 (-2^{15} to 2^{15}-1). A signed integer uses one bit for sign and 15 bits for magnitude of the number. Similarly, a 32 bit word length can store an integer ranging from -2147483648 to 2147483647 (-2^{31} to $+2^{31}$-1).

Ex1: int x =+100 (positive integer)



int y = -100(Negative Integer)



Unsigned Integer

Unsigned integer type variable allows only positive values. The Unsigned integers are using all 16 bits or 32 bits for storing the magnitude of a number. Here there is no sign bit because it does not allow negative values. If we use a 16 bit word length, the size of the unsigned integer value is limited to the range 0 to 65535 (0 to 2^{16}-1). A signed integer uses one bit for sign and 15 bits for magnitude of the number. Similarly, a 32 bit word length can store an integer ranging from 0 to 4294967295 (0 to $+2^{32}$-1).

Type	Key word	Size (bytes)	Range
Signed integer	int	2 (16 bits)	-32768 to 32767
Signed short integer	short int	2 (16 bits)	-32768 to 32767
Signed long integer	long int	4 (32 bits)	-2147483648 to 2147483647
Unsigned integer	unsigned int	2 (16 bits)	0 to 65535
Unsigned short integer	unsigned short int	2 (16 bits)	0 to 65535
Unsigned long integer	unsigned long int	4 (32 bits)	0 to 4294967295

15.1.2 *Character type*

A single character can be defined as a character (char) type data. Characters are usually stored in 8bits (one byte) of internal storage. The signed and unsigned characters both occupy one byte each but having different ranges.

The following table shows the size and range of character data types:

Type	Keyword	Size(byte)	Range
Singed character	Char or Singed char	1(8 bits)	-128 to 127
unsigned character	unsigned char	1(8 bits)	0 to 255

15.1.3 Floating point Data Type

Floating point data types are used to store real numbers. A Float occupies 4 bytes memory. If this is insufficient then C offers a double data type that occupies 8 bytes in memory. We can also use long double to store large real numbers. It occupies 10 bytes in memory.

The following table shows the size and range of Floating point data types:

Type	Keyword	Size (bytes)	Range	Precision for real
Float	float	4 (32 bits)	3.4 E–38 to 3.4 E+38	6 digits of precision
double	double	8 (64 bits)	1.7 E-308 to 1.7 E+308	15 digits of precision
long double	long double	10 (80 bits)	+3.4 E-4932 to 1.1 E+4932	provides between 16 and 30 decimal places

15.2 Derived data type

Derived data types such as arrays, functions, structures and pointers etc.

15.3 User defined data type

User defined data type are support some **type definition** and **enumeration**.

15.3.1 Type definition

The typedef is a keyword. By using **typedef** we can create new data type. The statement typedef is to be used while defining the new data type. The syntax is given below

typedef type dataname;

Here, *type* is the data type and *dataname* is the user defined name for that type.

typedef int hours;

Here, an hour is another name for int and now we can use hours instead of int in the program as follows.

hours hrs;

15.3.2 Enumeration

The enum is a keyword. It is used for declaring enumeration types. The programmer can create his/her own data type and define what values the variables of these data types can hold. This enumeration data type helps in reading the program.

Consider the example of 12 months of a year.

enum month {jan, feb, mar, apr, may, jun, jul, aug, sep, oct, nov, dec};

This statement creates a user defined data type. The keyword enum is followed by the tag name month. The enumeration is the identifiers jan, feb, mar, apr, and so on. Their values are constant unsigned integer start from 0. The identifier jan refers 0,feb refers 1 and so on.

15.4 Empty data set

Void is empty data set. The void type has no value. This is usually used to specify the type of the function. The type of the function said to be void when it does not return any value to the calling function.

16. Operators

C language provides a rich set of operators. They may operate on single operand or two operands. Operator is a symbol that tells the computer to perform certain mathematical or logical manipulation. An operator indicates an operation to be performed on data that yield a value. As operand is a data item on which operators perform the operation.

C operators can be classified into a number of categories. They include:

- ✓ Arithmetic operators
- ✓ Relational operators
- ✓ Logical operators
- ✓ Conditional operators
- ✓ Bitwise operators
- ✓ Assignment operators
- ✓ Comma operators

16.1 Arithmetic operators

There are two types of arithmetic operator. They are

1. Binary operators
2. Unary operator

16.1.1 Binary operators

These arithmetic operators commonly used for numerical calculations between the two constant values. The following table represents the binary operators.

Operator/Symbol	Description/Meaning	Example
+	Addition	5 + 4 = 9
-	Subtraction	7 – 3 = 4
*	Multiplication	6 * 2 = 12
/	Division	9 / 2 = 4
%	Modular division	7 % 3 = 1.

Modular division operator is used to find the reminder after an integer division.

Example Program

```
/*Binary Operators in c language*/
#include<stdio.h>
#include<conio.h>
main()
{
  int a,b;
  clrscr();
  printf("Enter a and b values");
  scanf("%d%d",&a,&b);
  printf("add-%d\nsub-%d\nmul-%d\ndiv-%d\nmod div-%d",a+b,a-b,a*b,a/b,a%b);
}
```

Output

```
Enter a and b values4 2
add-6
sub-2
mul-8
div-2
mod div-0
```

16.1.2 Unary Arithmetic Operators

The Unary operators can be applied on single operand only. The following table shows different Unary arithmetic operators that are used n C language:

Operator/Symbol	Description/Meaning
-	Unary Minus
++	Increment
_ _	Decrement
&	Address operator
Sizeof	Gives the size of operator

16.1.2.1 Unary Minus (-)

It is used to indicate or change the algebraic sign of a value.

For Example:

-125
int x= -50;
int y= -x;

16.1.2.2 Increment ++, Decrement – –

The compiler produces very fast, efficient object codes for increment and decrement operations. Increment or decrement operator is better than generated by using equivalent assignment statement.

The syntax of the operator is given below:

```
++<variable>          --<variable>
  <variable>++          <variable>--
```

The operator ++ adds 1 to the operand and -- subtract 1 from the operand. For example, X = X + 1 can be written as ++X or as X++. There is however a difference when they are used in expression.

The ++ and -- operators can be either in post-fixed or pre-fixed. A pre-increment operation such as ++a, increments the value of a by 1, before a is used for computation, while a post increment operation such as a++, uses the current value of a in the calculation and then increments the value of a by 1. Consider the following:

```
X = 10;
Y = ++X;
```

In this case, Y will be set to 11 because X is first incremented and then assigned to Y. However if the code had been written as

```
X = 10;
Y = X++;
```

Y would have been set to 10 and then X incremented. In both the cases, X is set to 11; the difference is when it happens.

16.1.2.3 Address Operator (&):

Ampersand (&) is referred as Address operator. It is usually precedes the identifier name, which indicates the memory allocation (address) of the identifier.

16.1.2.4 Sizeof Operator:

In situation where you need to incorporate the size of some object into an expression and also for the code to be portable across different machines, the size of unary operator will be useful.

The size of operator computes the size of any object at compile time. This can be used for dynamic memory allocation.

Usage: sizeof (object)

The object itself can be the name of any sort of variable or the name of a basic type (like int, float, char etc).

Example: sizeof (char) = 1 sizeof (int) = 2
sizeof (float) = 4 sizeof (double) = 8

Example Program

```
/* Unary Operator using C language */
#include<stdio.h>
#include<conio.h>
main()
{
  int x=-50,y;
  clrscr();
  printf("unary minus:%d",x);                    /* unary minus */
  y=8;     printf("\npre increment: %d",++y);    /* pre increment */
  y=8;     printf("\npost increment: %d",y++);   /* post increment */
  y=8;     printf("\npre decrement: %d",--y);    /* pre decrement */
  y=8;     printf("\npost decrement: %d",y--);   /* post decrement */
  printf("\naddress operator: %u",&y);           /* address operator */
  printf("\nsizeof operator: %d",sizeof(y));     /* sizeof operator */
}
```

Output

```
unary minus:-50
pre increment: 9
post increment: 8
pre decrement: 7
post decrement: 8
address operator: 65488
sizeof operator: 2
```

16.2 Relational Operators

The Relational operators are used to compare arithmetic, logical and character expressions. We often compare two similar quantities and depending on their relation, take some decisions. These comparisons can be done with the help of relational operators.

Each of these operators compares their left hand side with their right hand side. The whole expression involving the relational operator then evaluates to an integer. It evaluates to 1(One) if the condition is **true** and 0(zero) if it is **false**.

The following table shows different Relational operators that are used n C language:

Operator/Symbol	Description/Meaning	Example	Return Value
<	Less than	5 < 4	0
>	Greater than	7 > 3	1
<=	Less than or equal to	6 <= 6	1
>=	Greater than or equal to	9 >= 12	0
==	Equal to	10 == 9	0
!=	Not equal to	5 != 2	1

Example program

```
/* Relational Operator in C language */
#include<stdio.h>
#include<conio.h>
main()
{
 clrscr();
 printf("Less than: %d",5<4);                    /*Less than*/
 printf("\nGreater than: %d",7>3);                    /*Greater than*/
 printf("\nLess than or equal: %d",6<=6);          /*Less than or equal*/
 printf("\nGreater than or equal: %d",9>=12);      /*Greater than or equal*/
 printf("\nEqual to: %d",10==9);                        /*Equal to*/
 printf("\nNot equal to: %d",5!=2);                     /*not equal to*/
}
```

Output

```
Less than: 0
Greater than: 1
Less than or equal: 1
Greater than or equal: 0
Equal to: 0
Not equal to: 1
```

16.3 Logical Operators

The Logical operator is used to compare or evaluate logical or relational expressions. Using these operators, two expressions can be joined.

DEPARTMENT OF ELECTRICAL & ELECTRONICS ENGINEERING

An expression involving Logical AND(&&) or Logical OR(||) operator is sometimes called *compound expression*, since the expression involves two other expressions, that is, each of these operators(&& and ||) take two expressions, one to the left and another to the right.

The following table shows different Relational operators that are used n C language

Operator/Symbol	Description/Meanin	Example	Return Value
&&	Logical AND	5 > 4 && 3 < 6	1
||	Logical OR	7 > 3 || 15 <4	1
!	Logical NOT	!(9>4)	0

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

* The logical **AND (&&)** operator provides TRUE (1) result when both expressions are TRUE, otherwise FALSE (0).
* The Logical **OR (||)** operator provides TRUE (1) result when one of the expressions is TRUE, otherwise FALSE (0).
* The Logical **NOT (!)** operator provides FALSE(0) if the condition is TRUE and provides TRUE(1) if the condition is FALSE

The truth table for the logical operators is shown here using one's and zero's. In C true is any value other than zero (ie. Any Non-zero value), false is zero. Expressions is a relational or logical operators return zero for false and one for true.

P	Q	P && Q	P || Q	! P
0	0	0	0	1
0	1	0	1	1
1	0	0	1	0
1	1	1	1	0

Example program

```
/*Logical Operators in C language*/
#include<stdio.h>
#include<conio.h>
main()
{
 clrscr();
 printf("Logical AND: %d",5>4 && 3<6);        /*Logical AND*/
 printf("\nLogical OR: %d",7>3 || 15<4);       /*Logical OR*/
```

```
 printf("\nLogical NOT: %d",!(9>4));              /*Logical Not*/
}
```

Output

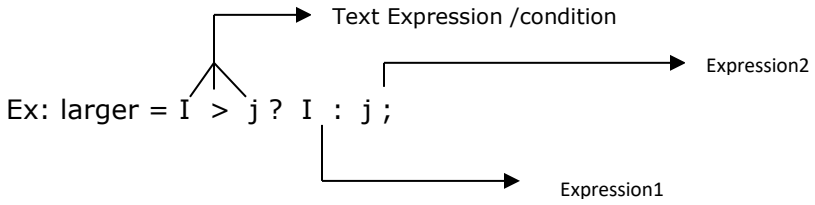
```
Logical AND: 1
Logical OR: 1
Logical NOT: 0
```

16.4 Conditional Operators (or Ternary operator)

The conditional operator consists of two symbols: the question mark(?) and colon(:). The conditional operator contains a condition followed by two statements or expressions. If the condition is TRUE the first expression or statement is executed. Otherwise the second expression or statement is executed. The conditional operator is also called as Ternary operator.

Syntax :

Condition ? expression1: expression2;



Example program

```
/*Conditional Operators in C language*/
#include<stdio.h>
#include<conio.h>
main()
{
 int i, j,larger;
 clrscr();
 i=10;
 j=20;
 larger= i>j?i:j; /* conditional operators(? :)*/
 printf("larger value among two numbers: %d",larger);
}
```

Output

```
larger value among two numbers: 20
```

16.5 Bitwise Operators:

A Bitwise operator operates on each bit of data. These operators are used for testing, complementing or shifting bits to the right or left. Usually bitwise operators are not useful in cases of float and double variables.

The ***bit wise*** operators of C are summarized in the following table:

Bitwise operators	
&	Bitwise AND
|	Bitwise OR
^	Bitwise XOR
~	One's Compliment
<<	Left shift
>>	Right Shift

The truth table for Bitwise operators AND, OR, and XOR is shown below. The table uses 1 for true and 0 for false.

P	**Q**	**P & Q**	**P | Q**	**P ^ Q**
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

The Bit wise AND (&) operator returns 1 if both the operands are one, otherwise it returns zero. For example, if y = 29 and z = 83, x = y & z the result is

```
0  0  0  1  1  1  0  1  →  29 in binary
                           &
0  1  0  1  0  0  1  1  →  83 in binary
0  0  0  1  0  0  0  1  →  Result (17)
```

The Bit wise OR (|) operator returns 1 if one or more bits have a value of 1, otherwise it returns zero. For example if, y = 29 and z = 83, x = y | z the result is:

0 0 0 1 1 1 0 1 → 29 in binary

|

0 1 0 1 0 0 1 1 → 83 in binary

0 1 0 1 1 1 1 1 → Result (95)

The Bit wise XOR (^) operator returns 1 if one of the operand is 1 and the other is zero, otherwise it returns zero. For example, if y = 29 and z = 83, x = y ^ z the result is

0 0 0 1 1 1 0 1 → 29 in binary

^

0 1 0 1 0 0 1 1 → 83 in binary

0 1 0 0 1 1 1 0 → Result(78)

One's complement (~) is a unary operator. It only operates on one operand, ie right of the operator. It finds 1's compliment (unary). It translates (or converts) all the 1 bits into 0's and all 0's into 1's

Example:

13 = 0000 0000 0000 1101

~13 = 1111 1111 1111 0010 = 65522

The shift operators (*Left shift* (<<) and *Right shift* (>>)) perform appropriate shift bits to the right or left of the operand. The right operator must be positive. The vacated bits are filled with zeros (***i.e.*** when shift operation takes places any bits shifted off are lost).

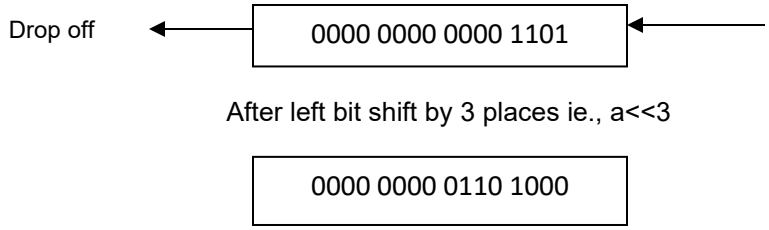
The *Left shift* (<<) operator is a binary operator which shifts the bit positions to the left. For example consider the statement

```
C=a << 3
```

The value in the integer 'a' is shifted to the left by three bit positions. The result is assigned to the integer 'c'. Since the value of 'a' is 0000 0000 0000 1101, then the value of 'c' after the

execution of the above statement is 0000 0000 0110 1000 (104 in decimal) and is illustrated below:

Left – Shift <<



The three left-most bits drop off due to the left shift. Three zeros are inserted in the right. The effect of shifting a variable to the left by one bit position is to *multiply it by 2*. Shifting bits to left means the decimal number is multiplied by 2s ie,.

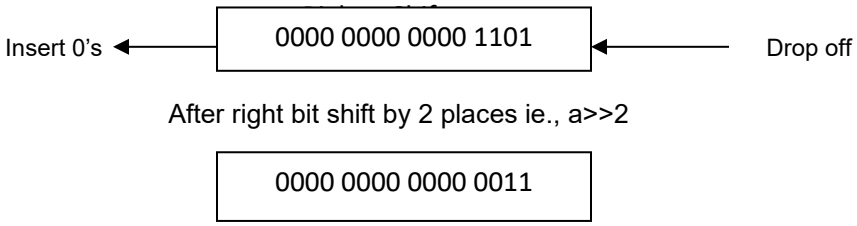
$$N * 2^s$$

Where 'N' is Number and's' is No. of bit positions to be shifted

The *Right shift* (>>) operator is also a binary operator which shifts the bit positions to the right. For example consider the statement

```
C=a >>2
```

The value in the integer 'a' is shifted to the right by two bit positions. The result is assigned to the integer 'c'. since the value of 'a' is 0000 0000 0000 1101 , then the value of 'c' after the execution of the above statement is : 0000 0000 0000 0011(3 in decimal) and is illustrated below:



The two right-most bits drop off due to the right shift. Two zeros are inserted in the right. The effect of shifting a variable to the right by one bit position is to perform integer *division by 2*. Shifting bits to left means the decimal number is divide by 2s ie,.

$$N / 2^s$$

Where 'N' is Number and's' is No. of bit positions to be shifted

NOTE Shifting is much faster than actual multiplication (*) or division (/) by 2.

Example program

```
/*Bitwise Operator in C language*/
#include<stdio.h>
#include<conio.h>
main()
{
 int y=29,z=83,a=13;
 clrscr();
 printf("Bitwise AND: %d",y&z);
 printf("\nBitwise OR: %d",y|z);
 printf("\nBitwise XOR: %d",y^z);
 printf("\nLeft Shift: %d",a<<3);
 printf("\nRight Shift: %d",a>>2);
 printf("\nOne's Complement: %u",~a);
}
```

Output

```
Bitwise AND: 17
Bitwise OR: 95
Bitwise XOR: 78
Left Shift: 104
Right Shift: 3
One's Complement: 65522
```

16.6 Assignment Operators:

Assignment operator (=) is used to Assign a value to a variable, either directly or as the result of a calculation.

Example

X = 10; Y= x; Z = x + y;

As we know this assignment operator(=) evaluates the expression on the right, and assigns the resulting value to the value on the left .Other forms of assignment operators exist, that are obtained by combining various operators such: + , -, *,/ and % etc., with the = sign.

For example, there is a += operator that evaluate the expression to its right, and adds the resulting value to the variable on its left.

The statement A+=5; will add the number 5 to the value of A. Similarly, A+= B – C will evaluate B – C first, and adds the result to the value of A. The other assignment operators are

Operator	Assignment operator	Shorthand operator
+ (addition)	a=a+b	a+=b

- (subtraction)	a=a-b	a-=b
* (multiplication)	a=a*b	a*=b
/ (division)	a=a/b	a/=b
% (modular division)	a=a%b	a%=b
& (bitwise AND)	a=a&b	a&=b
| (bitwise OR)	a=a|b	a|=b
^ (bitwise XOR)	a=a^b	a^=b
<< (left shift)	a=a<<b	a<<=b
>> (right shift)	a=a>>b	a>>=b

Note: there is no space between the operator and the '=' sign in compact representation. The operator = is known as short hand assignment operator.

The shorthand assignment operators are easy to read, write and understand and they also more efficient in evaluating an involved expression.

Example program

```
/*Assignment Operators in C language*/
#include<stdio.h>
#include<conio.h>
main()
{
 int a,b,c;
 clrscr();
 a=7; b=8; c=6; a=a*b+c;
 printf("Assignment Operator: %d",a);
 a=7; b=8;  c=6; a*=b+c;
 printf("\nShorthand Operator: %d",a);
}
```

Output

```
Assignment Operator: 62
Shorthand Operator: 98
```

16.7 Comma (,) Operator

Comma (,) operator is used to link the related expressions together. Comma used expressions are linked from left to right and the value of the right most expression is the value of the combined expression. The comma operator has the lowest precedence of all operators. For example:

(x = 12, y = 8, Sum = x + y;)

The result will be sum = 20.

The comma operator is also used to separate variables during declaration. For example:

```
int a, b, c;
```

Example program

```
/* Comma Operator in C language*/
#include<stdio.h>
#include<conio.h>
main()
{
 clrscr();
 printf("Addition= %d\nSubtraction=%d",2+5,6-5);
}
```

Output

```
Addition= 7
Subtraction=1
```

Example program 2

```
/*Comma Operator in C language*/
#include<stdio.h>
#include<conio.h>
main()
{
 int a=7;
 clrscr();
 printf("%3d%3d%3d%3d",a++,++a,a--,--a);
}
```

Output

```
 6  6  6  6
```

17. Operator precedence and Associativity

Every operator has a precedence value. This *precedence* is used to determine how an expression involving more than one operator is evaluated. There are distinct levels of precedence and the operators at the higher level of precedence are evaluated first.

DEPARTMENT OF ELECTRICAL & ELECTRONICS ENGINEERING

Associativity specifies the order in which the operators are evaluated with the same precedence in a complex expression. *Associativity* of an operator can be left-to-right or right-to-left. The following table shows the list operators with their precedence and associatity:

OPERATOR	Operation/Description	Precedence	Associativity
( ) [ ] -> .	Function Call Array subscript Structure Operator(Indirect selector) Structure Operator(Direct Selector)	1	Left to right
- + ++ -- ! ~ * & Sizeof (type cast)	Unary Minus Unary Plus Increment Decrement Logical Not operator Bitwise One's complement Pointer (Indirection)Operator Address Operator Size of Operator Type casting Operator	2	Right to Left
* / %	Multiplication Division Modular Division(Reminder)	3	Left to Right
+ -	Addition(Binary Plus) Subtraction(Binary Minus)	4	Left to Right
<< >>	Bitwise Left shift Bitwise Right Shift	5	Left to Right
< <= > >=	Less than Less than or Equal to Greater than Greater than or Equal to	6	Left to Right
== !=	Equal to Not Equal to	7	Left to Right
&	Bitwise AND	8	Left to Right
^	Bitwise XOR	9	Left to Right
|	Bitwise OR	10	Left to Right
&&	Logical AND	11	Left to Right
||	Logical OR	12	Left to Right
?:	Conditional Operator	13	Right to Left
=,+=,-+, *=, /=, %= ,etc.	Assignment Operators	14	Right to Left
,	Comma operator	15	Left to Right

18. Expressions and Expression Evaluation

An *Expression* is a combination of variables, constants and operators written according to the syntax of the language. In C, every expression evaluates to a value i.e., every expression results in some value of a certain type that can be assigned to a variable.

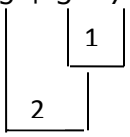
A Simple expression contains only one operator and a complex expression contain more than one operator.

Example: X = a + (b *c)/d

18.1 Expression Evaluation

Various operators have different priorities or precedence. If an arithmetic expression contains more operators, then the execution will be performed according to their priorities.

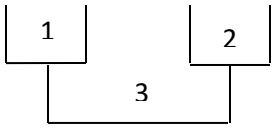
Example 1: x = 5 + 3 * 7



= 5 +21
X = 26

When two operators of the same priority or precedence are found in the expression, then the priority or precedence is given to the extreme left operator

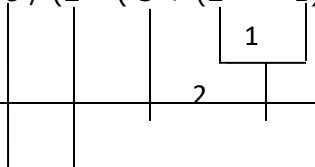
Example 2: x = 5 * 4 + 8 / 2

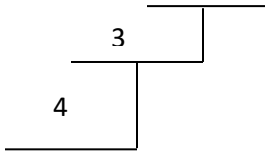


X = 20 + 4
X= 24

If more sets of parenthesis are in the expression, then the innermost parenthesis will be solved first, followed by the second and so on.

Example 3: X = 48 / (2 * (3 + (2 - 1)))





X= 48/(2*(3+1))
X= 48/(2 * 4)
X = 48/8
X = 6

19 Type Conversions

C language allows mixing of constants, and variables of different types in an expression. The *Type conversion* refers to different ways of changing an entity of one *data type* into another. There are two kinds of type conversions.

They are:

1. Implicit type conversion (or Automatic type conversion)
2. Explicit type conversion (or Type casting)

19.1 Implicit type conversion (or Automatic type conversion)

C automatically converts any intermediate values to the proper type , so that the expression can be evaluated without losing any significance. In this type conversion, the variable of lower type (which holds lower range of values or has lower precision) is converted a higher type (which holds higher range of values or has higher precision). This conversion is also called promotion.

Example 1:
```
int i=5;
float  f;
f = i;
```

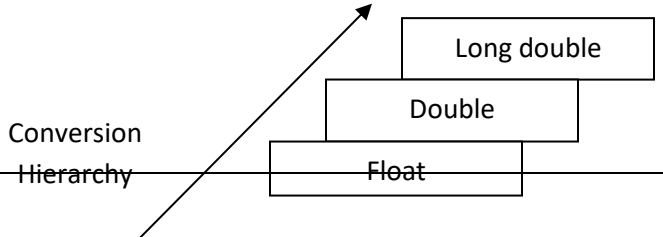
Here, the last statement assigns i to f. Since i is an integer while f is a floating point variable, the integer I is converted to float automatically.

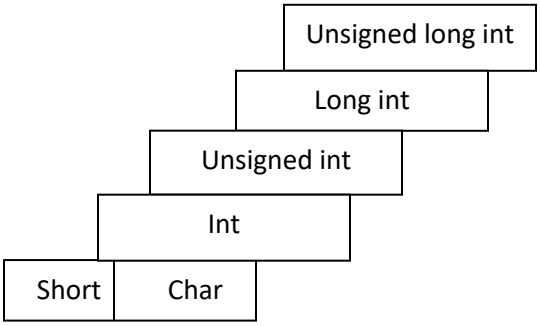
Example 2:
```
int i;
char c;
c='a';
i=c;
```

Here, the value present in the character variable, ie., the ASCII code of the character 'a' is assigned to the integer i. If an integer is represented by sixteen bits, the lower eight bits will have the value of this code, while the upper eight bits will be filled with zeros.

19.1.1 Conversion Hierarchy

C language uses the rule that, in all expressions except assignment, any implicit type conversions are made from a lower size type to a higher size type as shown below





Assignment of variables of higher types to those of lower type results in truncation

Example:
```
int i;
float f;
f = 7.5;
i=f;
```

Results in the fractional part (.5) discarded. The value 7 is assigned to the integer i.

19.2 Explicit Type conversion (or Type casting)

In addition to automatic conversions, we can forcibly convert one type to another. Explicit type conversion uses the Type casting operator to convert from one data type to another by specifying the type in parenthesis before the operand or expression to be converted. The general form or syntax is given below

(type) <variable> or (type) <expression>

Some of the expression of casts and their actions are shown in the following table:

Example	Action
X=(int) 7.5	7.5 is converted to integer by truncation and result would be 7.
A=(int) 21.3/(int) 4.5	Evaluated as 21/4 and result would be 5.
B=(float) 5/2	Division is done in floating point mode and result would be 2.5

Y= (int)(4.5 + 6.2)	The result of 4.5 + 6.2 is evaluated to integer ie., 10

13 Selection statements

(or Decision making/Conditional/Branching statements)

A program is nothing but the execution of sequence of one or more instruction. In some situations, it is necessary to check the condition to make a decision. This involves a performing a logical test. This test results in either a true or false. Depending upon the true or false of the condition, the statements to be executed are determined. After that, the control transfer to that statement in the program and starts executing the statement from that point. This is known as the conditional execution. The condition execution involves both decision making and branching. On the basis of applications it is essential.

- ✓ To alter the flow of a program.
- ✓ Test the logical conditions.
- ✓ Control the flow of execution as per the selection.

These conditions can be placed in the program using decision-making statement. C language supports the control statements such as

1. Simple if statement
2. if….else statement
3. Nested if statement
4. Switch statement

13.1 Simple if statement

C uses the keyword if to execute a set of statements or one statement when logical condition is true. It has only one option. It is also called a one-way branching. Here, the logical condition is tested which results in either a *true* or a *false* value.

If the result of the logical test is true (nonzero) then the statement that immediately follows *if* statement is executed. If the logical condition is false (zero), then control transfers to the next executable statement, outside the body of *if*. The condition must be written with in the parenthesis. The condition may be an expression containing constants, variables, or logical comparisons.

The syntax of an if statement is

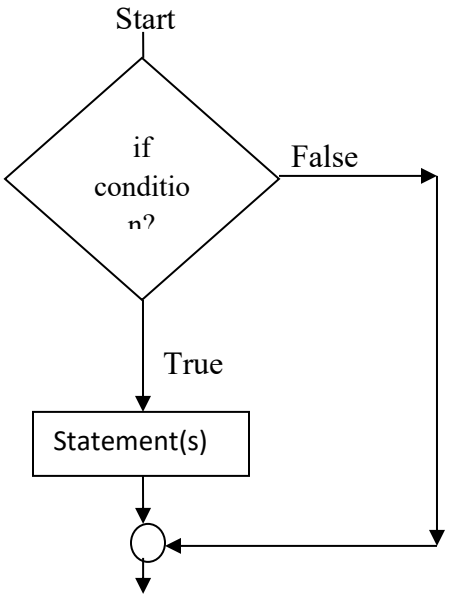
```
if(condition)
{
  Statement(s);
}
```

Where, condition → is a logical expression that results in **true** or **false**.

Statement → a sample or compound statement.

A simple statement is a single statement. On other hand, a compound statement is a collection of two or more statements placed between pair of braces. But, for a simple statement, no braces are required.

Flowchart



Exit or Next statement

Example program

Write a program to check whether the candidate is eligible for voting or not.

```
/* Example of simple if statement*/
#include<stdio.h>
#include<conio.h>
main()
{
 int age;
```

```
 clrscr();
 printf("Enter age:");
 scanf("%d",&age);
 if(age>=18)
  printf("Eligible for voting.");
}
```

Output

```
Enter age:20
Eligible for voting.
```

13.2 if….else statement

The *if-else* statement is used to execute only one action. If there are two statements to be executed alternatively, then *if-else* statement is used. The *if-else* statement is a two way branching.

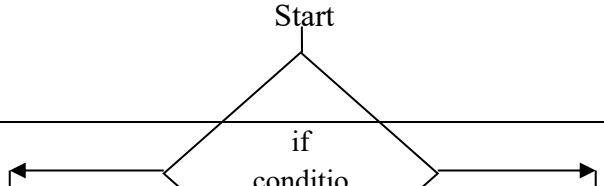
The syntax of if-else statement is

```
if (condition)
{
 Statement 1;
}
else
{
 Statement 2;
}
```

Or

```
if (condition)
{
  Statement1;
  Statement 2;
}
else
{
  Statement 3;
  Statement 4;
}
```

Flowchart



True False

Exit or Next Statement

The condition is tested and if the result of this logical test is true, then Statement 1 is executed. Otherwise, Statement 2 is executed. After executing one of these two statements, the control transfers to the statement immediately following the if-else structure. Here, statement1 and statement 2 may be simple or compound statement.

Or

The *if-else* statement takes care of true as well as false conditions. It has two blocks. One block is for *if* and it is executed when the condition is true. The other block is of *else* and it is executed when the condition is false. The *else* statement cannot be used without if. No multiple *else* are allowed with one if.

Example program

Write a program to check whether the given number is even or odd.

```
/* example of if....else statement*/
#include<stdio.h>
#include<conio.h>
main()
{
 int n;
 clrscr();
 printf("Enter number:");
 scanf("%d",&n);
 if(n%2==0)
  printf("Number %d is even.",n);
 else
  printf("Number %d is odd.",n);
}
```

Output

```
Enter number:4
Number 4 is even.
```

```
Enter number:5
Number 5 is odd.
```

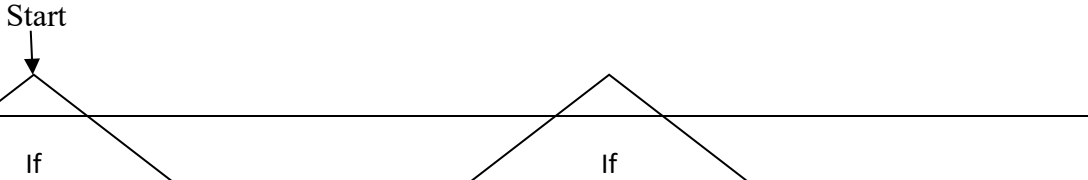
13.3 Nested if statement

In this kind of statements number of logical conditions is checked for executing various statements. Enclosing *if statement* with in another if statement is called *nested if statement*. Here, if any logical condition is true the compiler executes the block followed by *if* condition otherwise it skips and executes else block. In *if…else* statement *else* block is executed by default after failure of condition. In order to execute the *else* block depending upon certain condition we can add respectively if statements in *else* block.

The syntax of nested-if is

```
if (condition1)
{
   if(condition2)
   {
      Statement1;
   }
  else
   {
     Statement2;
   }
}
else if(condition3)
{
   Statement3;
 }
else
{
  Statement4;
}
```

Flowchart



In the above syntax, statement1 is executed if condition1 and condition2 are true. If condition1 is true and condition2 is false statement2 is executed. If condition1 is false then control transfers to the else-if part and tests condition3.

If condition3 is true then statement3 is executed. Otherwise, statement4 is executed. If condition1 and condition3 are false, the control comes out of this nested statement and continues with the next statement.

Example program

Write a C program to accept three integers and print the largest among them.

```
/*Example of nested if-else*/
#include<stdio.h>
#include<conio.h>
main()
{
 int a,b,c;
 clrscr();
 printf("Enter the values a,b and c:");
 scanf("%d%d%d",&a,&b,&c);
 if(a>b)
 {
  if(a>c)
   printf("a is the largest");
```

```
  else
   printf("c is the largest");
 }
else if(b>c)
{
    printf("b is the largest");
}
else
{
 printf("c is the largest");
}
}
```

Output

```
Enter the values a,b and c:2 3 4
c is the largest
```

```
Enter the values a,b and c:4 2 3
a is the largest
```

```
Enter the values a,b and c:2 4 3
b is the largest
```

if …else…if ladder

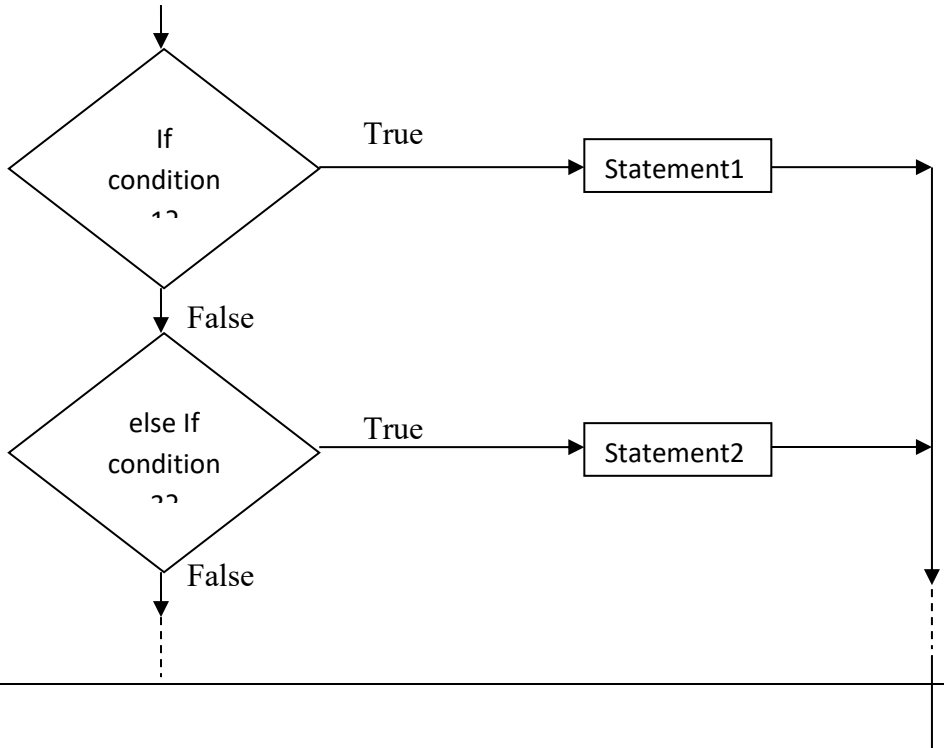
The below following rules can be described for applying nested if..else..if statement.

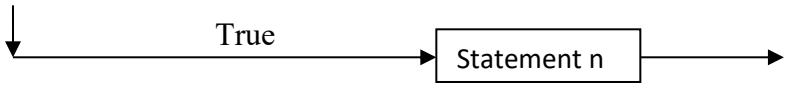
1. *Nested if…else* can be chained with one another.
2. if condition is false control *passes* to *else* block where condition is again checked with the *if* statement. This process continues if there is no *if* statement in the last *else* block.
3. if one of the *if* statement satisfies the condition, other nested *else…if* will not be executed.

The syntax of the if…else…if ladder is as follows

```
if(condition)
{
   Statement1;
   Statement2;
 }
else if(condition)
{
  Statement3;
  Statement4;
}
else
{
 Statement5;
 Statement 6;
}
```

Flowchart





Exit or next statement

Example program

Write a C program to accept three integers and print the largest among them.

```
/*Example of if-else-if ladder*/
#include<stdio.h>
#include<conio.h>
main()
{
 int a,b,c;
 clrscr();
 printf("Enter the values a,b and c:");
 scanf("%d%d%d",&a,&b,&c);
 if(a>b && a>c)
 {
   printf("a is the largest");
  }
 else if(b>a && b>c)
 {
   printf("b is the largest");
 }
 else
 {
   printf("c is the largest");
 }    }
```

Output

```
Enter the values a,b and c:2 3 4
c is the largest
```

```
Enter the values a,b and c:4 2 3
a is the largest
```

```
Enter the values a,b and c:2 4 3
b is the largest
```

13.4 Switch statement

The *if-else statement* provides a way for selecting any one of the two possible alternatives. And, the *nested-if* allows to select one of the many alternatives but it is time

consuming. It tests all conditions and based on the result a particular branch is taken for execution. To overcome that is, the *switch* statement is used. The switch statement provides a multiple way of branching. That is, it allows the user to select any one of the several alternatives, depending on the value of the expression. The expression is enclosed within the parenthesis.

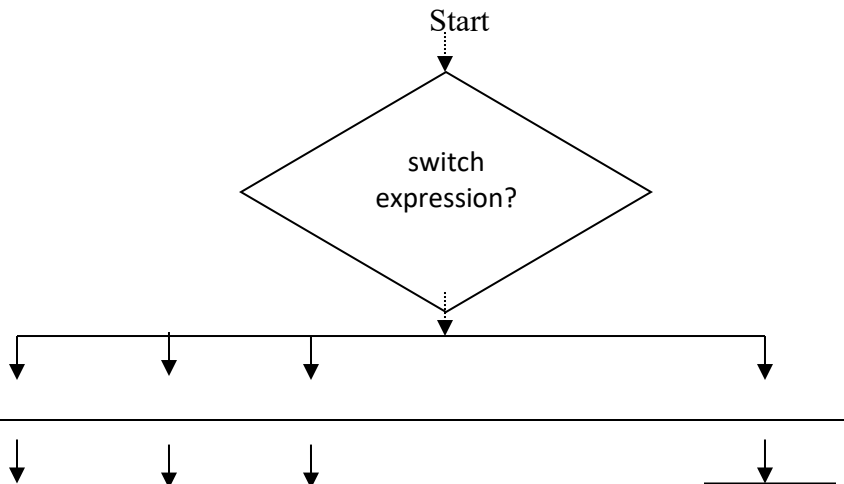
Depending upon value of the expression, the control transfers to a particular *case label (branch)* and all the statements followed by that *case label* are executed.

The syntax of *switch* statement is as follows

```
switch(expression)
{
 case label1: block1;
                break;
 case label2: block2;
        break;
 case label3: block3;
        break;
        .
        .
case labelmax: blockmax;
                break;
default: default block;
}
```

Here, the expression is of type int or char. Depending on the value of an expression, execution branches to a particular case label and then all the statements belonging to that case label are executed. The *break* statement indicates the end of a particular case label and thereby the switch statement is terminated. The *case default* is executed, when the value of an expression is not matched with any of the *case labels*. The semicolon should not be placed at the end of *switch (expression)*.

Flowchart



DEPARTMENT OF ELECTRICAL & ELECTRONICS ENGINEERING

case1 case2 case3 …… case n

Exit or next statement

Example program

Write a program to calculate the arithmetic operator using switch

```
/*Example of switch statement*/
#include<stdio.h>
#include<conio.h>
main()
{
 char ch;
 int a,b;
 clrscr();
 printf("Enter chioce(+,-,*,/,%):");
 scanf("%c",&ch);
 printf("enter the values a and b");
 scanf("%d%d",&a,&b);
 switch(ch)
 {
  case '+':printf("add-%d",a+b);
              break;
  case '-':printf("sub-%d",a-b);
              break;
  case '*':printf("mul-%d",a*b);
              break;
  case '/':printf("div-%d",a/b);
              break;
  case '%':printf("mod div-%d",a%b);
              break;
  default:printf("your chioce is wrong");
 }
}
```

Output

```
Enter chioce(+,-,*,/,%):*
enter the values a and b3 4
mul-12
```

17 Iteration (or) Loop control statement

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

Steps in loop

- ✓ *Loop variable* it is a variable used in the loop
- ✓ *Initialization* it is the first step in which starting and final value is assigned to the loop variable. Each time the updated value is checked by the loop itself.
- ✓ *Condition* an appropriate test condition to determine whether the loop to be execute or not.
- ✓ *Increment/decrement* it is the numerical value added or subtracted to the variable in each round of the loop

The C language supports three types of loop control statement.

1. While loop
2. do…while loop
3. for loop

17.1 while loop

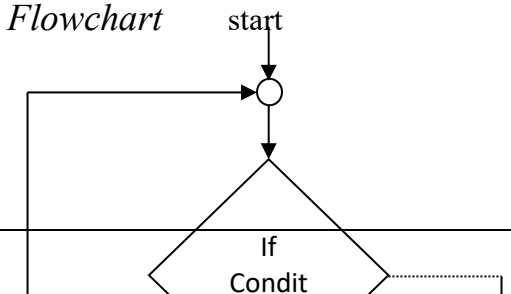
This is used to execute a set of statement repeatedly as long as the specified condition is true. The while loop control statement is an entry controlled statement. The test condition may be any expression. The loop statements will be executed till the condition is *true* i.e. the test condition is evaluated and if the condition is *true*, then the body of the loop is executed. When the condition becomes false the execution will be out of the loop.

```
Next - statement;
```

The syntax of while loop is given below

```
Initializing loop control variable;
while (condition)
{
  Body of the loop;
  Updating loop control variable;
}
```

Flowchart



False

True

Exit or next statement

Example program

Write a C program to sum of series (1+2+3+4+5+…..+n).

```
/*Example of while loop*/
#include<stdio.h>
#include<conio.h>
main()
{
 int n,sum=0,i=1;
 clrscr();
 printf("Enter the number:");
 scanf("%d",&n);
 while(i<=n)
 {
  sum=sum+i;
  i++;
 }
 printf("sum=%d",sum);
}
```

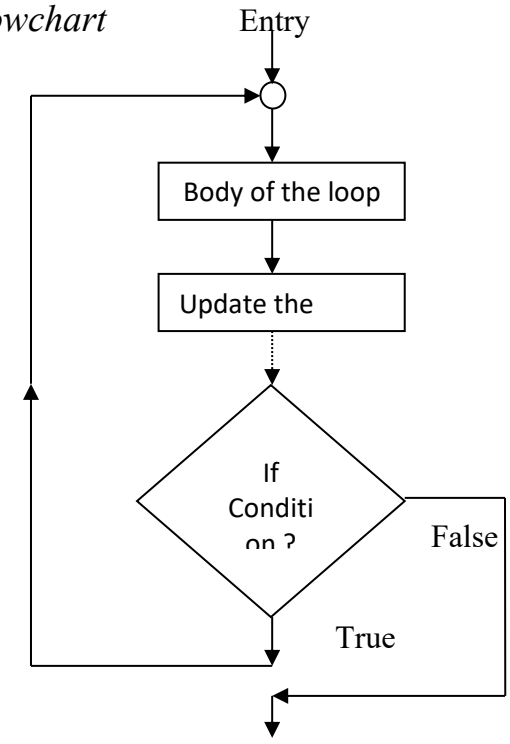
Output

```
Enter the number:5
sum=15
```

17.2 do….while

This is used to execute a set of statements repeatedly until the condition is false. The do….while statement is an exit controlled statement. The do – while loop checks its condition at the bottom of the loop. This means that the do – while loop always executes first and then the condition is tested. Unlike the while construction, the do – while requires a semicolon to follow the statement's conditional part.

Flowchart



The general format of do-while is:

```
Initializing loop control variable
do
  {
   Body of the loop;
   Update loop control variable
  }while (condition);
```

Example program

Write a C program to sum of series (1+2+3+4+5+…..+n) using do…while.

```
/*Example of while loop*/
#include<stdio.h>
#include<conio.h>
main()
{
 int n,sum=0,i=1;
 clrscr();
 printf("Enter the number:");
```

```
scanf("%d",&n);
do
{
 sum=sum+i;
 i++;
} while(i<=n);
printf("sum=%d",sum);
}
```

Output

```
Enter the number:5
sum=15
```

Note:

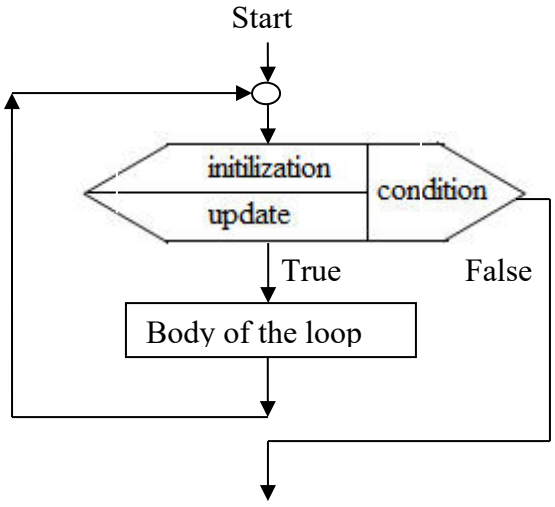
1. At least once the body of *do-while* is executed. Because the conditional test for repartition of the loop is carried out at the end of each pass.
2. The body of the *do-while* must contain either implicitly or explicitly statements to modify the variable involved in the *condition*.

17.3 for loop

The *for* loop control statement is also an entry controlled statement. *for* loop statement is useful to repeat a statement(s) a known number of times.

The general syntax is as follows:

```
for (initialization; condition; updation)
{
        Body of the loop;
}
Next-statement
```



- ✓ The **initialization** is generally an assignment statement that is used to set the loop control variable.

- ✓ The **condition** is an expression (relational/logical/arithmetic/bitwise ….) that determines when the loop exists.

- ✓ The **Updation (increment/decrement)** changes the loop control variable each time the loop is repeated.

We must separate these three major sections by semicolon.

for loop continues to execute as long as the condition is true. Once the condition becomes false, program execution resumes on the statement following *for*.

Example program

Write a C program to sum of series (1+2+3+4+5+…..+n) using do…while.

```
/*Example of while loop*/
#include<stdio.h>
#include<conio.h>
main()
{
 int n,sum=0,i;
 clrscr();
 printf("Enter the number:");
 scanf("%d",&n);
 for(i=0;i<=n;i++)
 { sum=sum+i;
 }
 printf("sum=%d",sum);
}
```

Output

```
Enter the number:5
sum=15
```

Important things to know about for loop

- The **comma (,) operator** is used to extend the flexibility of the for loop. It allows the general form to be modified as follows:

```
for (initialization_1, initialization_2; condition; updation_1,updation_2)
{
        statement;
}
```

- All sections of a for loop is optional(you may use or may not use).Then the loop repeat for infinite times. We can make an endless loop (*Infinite for loop*) by leaving the conditional expression empty as given below:

for(; ;) This loop will run for ever

To terminate the infinite loop the break statement can be used anywhere inside the body of the loop. A sample example is given below:

```
for(; ;)
{
        ch = getchar ();
        if(ch == 'A')
                break;
}
printf("You typed an A");
```

This loop will run until the user types an A at the keyboard.

- Conditional section of a for loop can have multiple condition also but the right most condition will take the decision.

Example program

```
/*Example of for loop in comma*/
#include<stdio.h>
#include<conio.h>
main()
{
 int n,i,j;
 clrscr();
 printf("Enter the number:");
 scanf("%d",&n);
 for(i=1,j=1;i<=n;i++,j++)
 {
  printf("%d %d\n",i,j);
```

```
 }
}
```

Output

```
Enter the number:5
1 1
2 2
3 3
4 4
5 5
```

Nested for loop

A **for** loop placed inside another **for** loop is called nested **for** loop.

General form of nested for loop is:

```
for (initialization; condition; operation)
{
        for (initialization; condition; operation)
        {
                statement;
        }
        statement;
}
```

In this syntax, the inner loop runs through its full range of iterations for each single iteration of the outer loop.

Example program

```
/*Example of for loop in comma*/
#include<stdio.h>
#include<conio.h>
main()
{
 int n,i,j;
 clrscr();
```

```
 printf("Enter the number:");
 scanf("%d",&n);
 for(i=1;i<=n;i++)
 {
  for(j=1;j<=n;j++)
  {
  printf("%d %d\n",i,j);
  }
 }
}
```

Output

```
Enter the number:3
1 1
1 2
1 3
2 1
2 2
2 3
3 1
3 2
3 3
```

. Break statement

The keyword *break* allows the programmers to terminate the loop. The *break* skips from the loop or block in which it is defined. The control then automatically goes to the first statement after the loop or block. The break can be associated with all conditional statement.

We can also use *break* statements in the nested loops. If we use *break* statement in the inner loop then the control of the program is terminated only from the innermost loop.

Example program

Write a C program to 1 to n prime numbers using break statement.

```
/*find the 1 to n prime numbers*/
#include<stdio.h>
#include<conio.h>
main()
{
 int i,j,p,n;
 clrscr();
 printf("Enter n:");
 scanf("%d",&n);
```

```
for(i=2;i<=n;i++)
{
  p=1;
  for(j=2;j<i;j++)
  {
   if(i%j==0)
   {
   p=0;
   break;
   }
  }
 if(p)
  printf("%d ",i);
}
}
```

Output

```
Enter n:10
2 3 5 7
```

6. Continue statement

The continue statement is exactly opposite to break. The continue statement is used for continuing next iteration of loop statements. When it occurs in the loop it does not terminate, but it skips the statements after this statement. It is useful when we want to continue the program without executing any part of the program.

Example program

Write a C program to 1 to n even numbers using continue statement.

```
/* find 1 to n even numbers*/
#include<stdio.h>
#include<conio.h>
main()
{
 int i,n;
 clrscr();
 printf("Enter n:");
 scanf("%d",&n);
 for(i=1;i<n;i++)
 {
  if(i%2!=0)
  {
   continue;
```

```
 }
 printf("%d ",i);
 }
}
```

Output

```
Enter n:10
2 4 6 8
```

7. goto statement

C language supports the unconditional control statement, goto, to transfer the control from one point to another in a C program. The *goto* is a branch statement and requires a *label*.

The syntax of goto statement
goto label;
Where, goto → is a keyword
Label → is a symbolic constant written either in uppercase or lowercase letters.
The label can be placed anywhere in the C program either before or after the goto statement.
For example:
Consider the following two program segment

```
goto END;
…….
…….
…….
END:
…….
```

(a)

```
START:
…….
…….
…….
goto START;
```

(b)

In (a) the label END is placed after the goto END; statement. Here, the statement immediately followed by the goto will be skipped, while the control jumping to the label END. this type of jumping is called **forward jump**.

In(b) the label START: is placed before the goto statement. This type of jump is known as a **backward jump** and it will form a loop. The statements inside this loop will be executed repeatedly.

Example program

Write a C program to given number is even numbers or odd number using goto statement.

```
/*Example of forward jump*/
#include<stdio.h>
#include<conio.h>
main()
{
 int n;
 clrscr();
 printf("Enter n:");
 scanf("%d",&n);
 if(n%2==0)
 {
  goto even;
 }
 else
 {
  goto odd;
 }
 even: printf("even %d",n);
          return0;
 odd: printf("odd %d",n);
}
```

Output

```
Enter n:5
odd 5
```

```
Enter n:4
even 4
```

Example program

Write a C program to sum of natural numbers using goto.

```
/*Example for backward jump*/
#include<stdio.h>
#include<conio.h>
main()
{
 int n,sum=0,i=1;
 clrscr();
 printf("enter the number:");
```

```
scanf("%d",&n);
add:
    sum=sum+i;
    i++;
    if(i<=n)
goto add;
printf("sum=%d",sum);
}
```

Output

```
enter the number:5
sum=15
```

Arrays

10 Definition of an array

An array can be defined as an ordered list of homogeneous data element. (or) *an array is a collection of similar data types in which each element is unique one and located into separate memory location*. These elements may be of type int, float, char, or double. All these elements are stored in consecutive memory locations (on RAM).

An array is described by a single name or an identifier. And each element in an array is referenced by a *subscript* (or index) enclosed in a pair of square brackets. This subscript indicates the position of an individual data item in an array. The subscript must be an unsigned positive integer. Because of these subscripts, sometimes an array is called as a *subscript variable*.

10.1 Rules for subscript

- Some of the rules that govern the subscripts are given below.
- Each subscript must be an unsigned positive integer constant or expression.
- Subscript of subscript is not allowed.
- C does not perform bound checking. Therefore, the maximum subscript appearing in a program for a subscripted variable should not exceed the declare one.
- In C, the subscript value range from **0** to one less than the **maximum size**. If the size of an array is 10, the first subscript is 0, the second subscript is 1 and so on the last subscript is 9. In general, the i^{th} element has the subscript (i-1).
- For example, consider an array **list** having 5 elements. The subscript that denotes the position of an individual item is as shown below

Element location	list[0]	list[1]	list[2]	list[3]	list[4]
Address	2000	2002	2004	2006	2008

The elements of an integer list[5] are stored in continuous memory locations. It assumed that the starting memory location is 2000. Each integer element requires 2 bytes. Hence subsequent element appears after gap of 2 locations.

11 Classification of array

Arrays are classified into two types: One dimensional array and multi dimensional arrays, further, the multi dimensional arrays are classified into a two dimensional, a three dimensional and soon n dimensional array.

The dimensionality of an array is determined by the number of subscripts present in the given array. If there is only one subscript, then it is called a one-dimensional array. If there are two subscripts, it is called a two-dimensional array and so on.

11.1 One dimensional array

This is a linear list of a fixed number of data items of the same data type. All these data items are accessed using same name and single subscript is called *single dimensional array* or *a one subscript array.*

11.1.1 Declaration of a one-dimensional array

The syntax of declaring a one-dimensional is as follows

```
Data  type    arrayname[50];
```

Where,

Data_type → any basic data type or a user-defined data type.
arrayname → is the name of the array
size → number of elements of type *data_type*. And the size must be an integer constant specified with in a pair of square brackets

Examples:

1. int list[10];
2. char name[20];
3. float xyz[5];
4. double p[100];

The first declaration creates an array named list of 10 integer constants. The second declaration create name as an array of 20 characters. The third declaration creates xyz as an array of 5 floating-point numbers. Finally, the fourth declaration creates p as an array of 100 double precision numbers.

Total bytes

The total amount of memory that can be allocated to a one-dimensional array is computed as,

```
Total bytes = sizeof(data_type) * size;
```

Where,

Size → is a number of elements of a one-dimensional array
Sizeof() → is an unary operator to find the size in bytes
Data_type → basic data type or a user defined data type

For example,

The total bytes of one-dimensional array of 10 integers are 20 bytes. The size of an integer data is 2 bytes.

11.1.2 Initializing a one-dimensional array

Initializing is a process of assigning value to a variable that undergoes processing, like initialization to an ordinary variable, the individual elements of an array can also be initialized. All these initial values must be constants. But they can never be variables or function calls. Initialization can be made to an all array elements during the time of declaration.

The syntax for one-dimensional array initialization is as follows:

```
data_type   arrayname[size]={element1,element2,……..,elenemtn};
```

Where,

data_type → basic data type or a user defined data type.

arrayname → name of array.

size → maximum number of elements in the array.

element1, element2,…….elementn →are the initial values enclosed with in a pair of curly braces. And all these elements must be separated by a comma. These elements must be written in the order in which they will be assigned.

Example:

```
int even[4]={2,4,6,8};
```

Element location	even[0]	even[1]	even[2]	even[3]
Element (value)	2	4	6	8
Address(memory)	2000	2002	2004	2008

Array elements are called by array names followed by the element numbers.

even[0] refers to 1st element i.e. 2

even[1] refers to 2nd element i.e. 4

even[2] refers to 3rd element i.e. 6
even[3] refers to 4th element i.e. 8

11.2 Processing of an Array (Using for Loops for Sequential Access)

Very often, we wish to process the elements of an array in sequence, starting with element zero. An example would be scanning data into the array or printing its contents. In C, we can accomplish this processing easily using an *indexed* **for** loop, a counting loop whose loop control variable runs from zero to one less than the array size.

Using the loop counter as an array *index* (subscript) gives access to each array element in turn (reading & printing array by using for loop). One common use of arrays is for storage of a collection of related data values

Reading an array

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

in the above for loop using three processing steps the array **a**. one is the initial value is 0, condition is used and incrimination. Stores the one input value into each element of array **a** (the first item is placed into a[0], the second item is placed into a[1], an soon). The call to *scanf* is repeated for each value of i from 0 to size.

For example, the initial value of **i** is 0, if the **size** is 5, the condition is satisfied up to less than 5 and increment the value of **i**. each repetition gets a new data value and stores it in a[i]. The subscript **i** determines which array element receives the next data.

Printing an array

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

in the above for loop using three processing steps the array **a**. one is the initial value is 0, condition is used and incrimination. Print the value on the console one output value into each element of array **a**.

Example program

Write a program to initialize and print the one-dimensional array values

```
/*array elements initialization and printing*/
#include<stdio.h>
#include<conio.h>
main()
{
 int a[ ]={1,2,3,4,5},i;
 clrscr();
 /*printing an array element*/
 printf("Print the array element:");
 for(i=0;i<5;i++)
 {
  printf("%3d",a[i]);
 }
}
```

Output

```
Print the array element:  1 2 3 4 5
```

Example program

Write a program to read and print the array values.

```
/*array elements reading and printing*/
#include<stdio.h>
#include<conio.h>
main()
{
 int a[50],i,n,sum=0;
 clrscr();
 printf("Enter n");
 scanf("%d",&n);

 /*reading of array element*/
 printf("Enter array values");
```

```
 for(i=0;i<n;i++)
 scanf("%d",&a[i]);
/
*printing an array element*/
 printf("Print the array element:");
 for(i=0;i<5;i++)
 {
  printf("%3d",a[i]);
 }
}
```

Output

```
Enter n: 5
Enter array values: 1 2 3 4 5
Print the array element:  1 2 3 4 5
```

Example program

Write a program to find the sum of all elements in an array

```
/*sum of array elements*/
#include<stdio.h>
#include<conio.h>
main()
{
 int a[50],i,n,sum=0;
 clrscr();
 printf("Enter n:");
 scanf("%d",&n);

 /*reading of array element*/
 printf("Enter array values:");
 for(i=0;i<n;i++)
 scanf("%d",&a[i]);
 /*caluclate array elements*/
 printf("Sum of array elements:");
 for(i=0;i<n;i++)
 {  sum=sum+a[i];
```

```
 }
 printf("%d",sum);
}
```

Output

```
Enter n:5
Enter array values:1 5 2 4 3
Sum of array elements: 15
```

Example program

Write a program to find the product of all elements in an array
/*product of array elements*/

```
#include<stdio.h>
#include<conio.h>
main()
{
 int a[50],i,n,prod=1;
 clrscr();
 printf("Enter n:");
 scanf("%d",&n);
 /*reading of array element*/
 printf("Enter array values:");
 for(i=0;i<n;i++)
 scanf("%d",&a[i]);
 /*caluclate array elements*/
 printf("Product of array element:");
 for(i=0;i<n;i++)
 {  prod=prod*a[i];
 }
 printf("%d",prod);
}
```

Output

```
Enter n: 5
Enter array values: 2 1 3 5 4
Product of array element: 120
```

11.3Multi dimensional arrays

If the number of subscripts more than one then such arrays are called *multi dimensional arrays*. Thus, we can have a two-dimensional (2-D) array, three-dimensional (3-D) array and soon. The dimensionality is determined by the number of pairs of square brackets placed after the array name.

Example

1. array1[ ] → one-dimensional array
2. array[ ][ ] →two-dimensional array
3. array[ ][ ][ ] →three-dimensional array

The programmer will fill in each pair of brackets with the number of elements to be processed.

11.3.1Two-dimensional array

It is an ordered table of homogeneous elements. It is generally, referred to as a matrix of some rows and some columns. It is also called as a two-subscripted variable.

Declaration of a two-dimensional array

The syntax of declaring a two-dimensional array in C is as follows

```
data type arrayname[rows][columns];
```

Where,

rows →number of elements to be processed under *subscript1*.
columns →number of elements to be processed under *subscript2*.

Example

1. int marks[5][3]
2. float matrix[3][3]
3. char page[25][80]

The first example specifies that the **marks** are a two-dimensional array of 5 rows and 3 columns. Rows may represent *students* and the columns the *tests*. The second example indicates that the **matrix** is a two-dimensional array of 3 rows and 3 columns. Similarly, the example declares a **page** as a two-dimensional array of 25 rows and 80 columns.

Two-dimensional array can be thought as a rectangular display of elements with rows and columns. For example elements of int x[3][3] are shown as below.

	Col 0	Col 1	Col 2
Row 0	x[0][0]	x[0][1]	x[0][2]
Row 1	x[1][0]	x[1][1]	x[1][2]
Row 2	x[2][0]	x[2][1]	x[2][2]

Conceptually the elements are shown in matrix form. Physically array elements are stored in one continues form in memory. The two-dimensional array is a collection of a number of one-dimensional arrays, which are placed one after another.

Initialization of a two-dimensional array

Like one-dimensional array elements, the two-dimensional array elements can also be initialized at the time of declaration. The syntax for initializing a two –dimensional array is as follows:

```
data_type   arrayname[size1][size2]={e1, e2, e3…….....en};
```

Where,

data_type → basic data type.
array_name → name of a two-dimensional array.
e1,e2,e3……en → initial values to be assigned to n elements of an array.
size1, size2 → maximum number of rows and columns.

Example 1: consider the following declaration,
int x[3][3]={1, 2, 3, 4, 5, 6, 7, 8, 9};
Then, the first 9 elements of the **matrix** will be,
Memory map of two dimensional array elements

row, col	x[0][0]	x[0][1]	x[0][2]	x[1][0]	x[1][1]	x[1][2]	x[2][0]	x[2][1]	x[2][2]

Value	1	2	3	4	5	6	7	8	9
Memory	2000	2002	2004	2006	2008	2010	2012	2014	2015

If the number of elements to be assigned is less than the total number of elements that a two-dimensional array has contained, then all the remaining elements of an array are assigned to zeros.

int x[2][2]={{1, 2}, {3, 4}};

Here, the inner set {1,2} is taken for assigning values 1 and 2, respectively, to the first row elements of an array x. thus,

x[0][0]=1 x[0][1]=2

Similarly, the other set {3,4} is taken for assigning its values 3 and 4, respectively, to the second row element of an array x. Thus,

x[1][0]=3 x[1][1]=4

Remember the following points while dealing with the initialization of the two-dimensional array elements.

1. The number of set of initial values must be equal to the number of rows in the arrays.
2. One to one mapping is preserved. i.e. the first set of initial value is assigned to the first row elements and the second set of initial values is assigned to the second row elements and so on.
3. If the number of initial values in each initializing set is less than the number of the corresponding row elements, then all the elements of that row are automatically assigned to zeros.
4. If the number of initial values in each initializing set exceeds the number of the corresponding row elements then there will be a compilation error.

Example program

Write a program to initialize and print the two-dimensional array values.

```
/*array elements initilization & printing*/
#include<stdio.h>
#include<conio.h>
main()
```

```
{
 int a[5][5]={{1,2,3},{4,5,6},{7,8,9}},i,j;
 clrscr();
 printf("print the two dimentional array element:\n");
 for(i=0;i<3;i++)
 {
  for(j=0;j<3;j++)
  {
  printf("%3d",a[i][j]);
  }
 printf("\n");
 }
}
```

Output

```
print the two dimentional array element:
  1  2  3
  4  5  6
  7  8  9
```

Processing of two dimensional array

A two-dimensional array is generally called a matrix. Therefore, a certain kind of operations can be performed on matrices using two-dimensional array.

1. *Reading and printing of elements*
2. *Adding and subtraction the corresponding elements of two matrices.*
3. *Multiplication of two matrices and etc.*

Example program:

Write a program to reading and print the two-dimensional array values.

```
/*array elements reading & printing*/
#include<stdio.h>
#include<conio.h>
main()
```

```
{
    int a[5][5],i,j,m,n;
    clrscr();
    printf("Enter row(m) & col(n):");
    scanf("%d%d",&m,&n);
    /*reading two dimentional array*/
     printf("Enter matrix:");
    for(i=0;i<m;i++)
    {
            for(j=0;j<n;j++)
            {
             scanf("%d",&a[i][j]);
            }
    }
    /*printing two dimentional array */
    printf("Print the Matrix:\n");
    for(i=0;i<m;i++)
    {
             for(j=0;j<n;j++)
            {
            printf("%3d",a[i][j]);
             }
    printf("\n");
    }
}
```

Output:

```
Enter row(m) & col(n):3 3
Enter matrix:
1 2 3
4 5 6


7 8 9
Print the Matrix:
  1  2  3
```

```
4 5 6
7 8 9
```

11.3.2 Three- or multi dimensional arrays

The c program allows array of two or multi dimensions. The compiler determines the restriction on it.

Declaration of three-or multi dimensional arrays

The syntax of multi-dimensional array as follows.

```
data  type   arrayname[s1][s2][s3]…….[sn];
```

Where,

s1→number of elements to be processed under *subscript1*.
s2 →number of elements to be processed under *subscript2*.
s3→ number of elements to be processed under *subscript3*. ……So on.
sn→ number of elements to be processed under *subscript n.*

Example:

1. int mat[3][3][3];
2. float m[3][3][3];
3. char s[9][9][9]

Initialization of three-or multi dimensional arrays

. The syntax for initializing a two –dimensional array is as follows:

```
data_type   arrayname[s1][s2][s3]……[sn]={e1, e2, e3…..........en};
```

Where,

data_type → basic data type.
array_name → name of a two-dimensional array.
e1,e2,e3……en → initial values to be assigned to n elements of an array.
s1, s2, ….sn → maximum number of rows and columns.

Example:

```
int mat[2][2][2]={1, 2, 3, 4, 5, 6,7, 8};
                or
int mat[2][2][2]={{1, 2, 3, 4},{ 5, 6,7, 8}};
```

A three dimensional array can be thought of as an array of arrays. The outer array contains three elements. The inner array size is two dimensional with size [2][2].

Example program

Write a program to initialization and print the three-dimensional array values.

```
/*Three-dimensional array initialization & printing*/
#include<stdio.h>
#include<conio.h>
main()
{
 int m[2][2][2]={{1,2,2,3},{3,4,4,5},},i,j,k;
   clrscr();
   /*printing an array element*/
  printf("Print the array elements are:");
   for(i=0;i<2;i++)
   {
    for(j=0;j<2;j++)
    {
     for(k=0;k<2;k++)
     {
      printf("%d",m[i][j][k]);
     }
    }
   }
 }
```

Output:

```
Print the array elements are:1 2 2 3 3 4 4 5
```

Strings

12 Definition of the string

In C language the group of characters, digits, and symbols enclosed within quotation marks called as String. The string is always declared as character arrays. In other words character arrays are called String.

Every string terminated with '\0' (NULL) character. The NULL character is a byte with all bits at logic zero. Hence, its decimal vale is zero.

For example

char name[ ]={'M','U','R','A','H','A','R','I','\0'};

Each character of the string occupies 1 byte of memory. The last character always '\0'. It is not compulsory to write '\0' in string. The Compiler automatically puts '\0' at the end of the character array or string. The characters of string are stored in contiguous (neighboring) memory locations.

Element	M	U	R	A	H	A	R	I	\0
Memory	5000	5001	5002	5003	5004	5005	5006	5007	5008

12.1 Declaration and initialization of string

The string a group of characters, digits and special symbols enclosed within the double quotation marks. The string always ends with '\0'.

Syntax for declaring and initializing the string is as follows.

char name[ ]="string"; or char name[ ]={ element of string};

char → it is a basic data type.
name → array of the string name.

Example of the string

char a[ ]="murahari";

The C compiler inserts the NULL (\0) character automatically at the end of the string. So initialization of NULL character is not essential.

Also,

Character arrays can be initialized as follows.

char a[8]={'m','u','a','r','h','a','r','i'};

12.2 Reading a string from the keyboard

Use of scanf() function

The familiar input function scanf() can be used with %s format specification to read in a string of characters. For example

```
Char name[10];
Scanf("%s",name);
```

The problem with the scanf( ) function is that it terminates its input on the first white space it finds. Unlike previous scanf calls, in the case of character arrays, the ampersand(&) is not required before the variable name.

Use of gets() function

The library functions gets() is a easiest and more convenient method of reading a string of text containing whitespaces input a string from the keyboard is with the gets() library function which allows a st. The general form gets() is:

```
gets(array_name);
```

For example :

```
Char name[10];
Gets(name);
```

To read a string, call gets() with the name of the array, with out any index, as its arguments. The gets() function will continue to read characters until you enter a carriage return. The header file used for gets() is stdio.h

Example

```
# include <stdio.h>
main()
{
    char str[80];
    printf ("\nEnter a string:");
    gets (str);
    printf ("%s", str);
}
```

The carriage return does not become part of the string instead a null terminator is placed at the end.

12.3 Printing or Writing strings to screen

Using printf() function

We can use printf() function with %s format to print strings to the screen.

For example :

```
Printf("%s",name);
```

Note that printf( ) doesn't print the '\0' on the screen.

```
main( )
{
 char  city[ ] = "Hyderabad" ;
 printf ( "%s", name ) ;
}
```

Using Puts() function

The puts() functions writes its string argument to the screen followed by a newline. Its prototype is:

```
puts(string);
```

For example, to display "hello" on the screen:

```
puts("hello");
```

13. Array of strings

To create an array of strings, a two dimensional character array is used with the size of the left-Index determining the number of strings and the size of the right Index specifying the maximum length of each string.

For example, to declare an array of 10 strings each having a max length of 30 characters.

```
char name[10][30];
```

The order of the subscripts in the array declaration is important. The first subscript gives the number of names in the array, while the second subscript gives the length of each item in the array.

To access an individual string is quite easy: you simply specify only the left Index. The array of strings can be initialized as shown below:

```
Char months[12][30] = { "January", " February  " ,
                        "March", "april" , "May",
                        "June", "July",
                        "Agust", " September",
                        "October", November"
                        "December"
                      };
```

14. String Library Functions (or String Handling Functions)

C- supports a large number of string handling functions that can be used to carry out many of the string manipulations. Following are the most commonly used string handling or string library functions

S.No.	Name	Function
1	strcpy(s1, s2)	Copies s2 into s1 (the array forming to must be large enough to hold the string content in form)
2	strcat(s1, s2)	Concatenates s2 onto the end of s1
3	strlen(s1)	Returns the length of s1
4	strcmp(s1, s2)	Returns 0 if s1 and s2 are the same to determine alphabetic order. Less than 0 if s1 < s2; greater than 0 if s1 > s2
5	strchr(s1,ch)	Return a pointer to first occurrence of *ch* in *s1*
6	strstr(s1,s2)	Return a pointer to first occurrence of *s2* in *s1*
7	strrev(s1)	Reverses the string s1.
8	Strlwr(s1)	Converts a string s1 to lowercase
9	strupr (s1)	Converts a string s1 to uppercase
10	strset(s1,#)	

All the string handling functions are prototyped in: # include <string.h>

Example program

```
/* String handling functions */
#include<stdio.h>
#include<string.h>
main()
{
 char s1[80],s2[80];
 clrscr();
 printf("enter s1:");
 gets(s1);
 printf("enter s2:");
 gets(s2);
 printf("length of s1:%d\n",strlen(s1));
 if(!strcmp(s1,s2))
  printf("the strings are equal\n");
 strcat(s1,s2);
 printf("s1=%s\n",s1);
 strcpy(s1,"this is a test.");
 puts(s1);
```

```
 if(strchr("hello",'e'))

printf("e is found in hello\n");
 if(strstr("hi there","hi"))
      printf("found hi\n");
  printf("strrev=%s\n",strrev("hari"));
  printf("strupr=%s\n",strupr("hari"));
  printf("strlwr=%s\n",strlwr("HARI"));
  printf("strset=%s\n",strset("hari",'#'));
}
```

Output

```
enter s1:hello
enter s2:hello
length of s1:5
the strings are equal
s1=hellohello
this is a test.
e is found in hello
found hi
strrev=irah
strupr=HARI
strlwr=hari
strset=#####
```

UNIT-III

Functions

1. Definition of the function

A function is a self contained block or a sub program of one or more statement that perform a special task when called.

Types of Functions

The c language support two types of functions

1. Library function (standard function)
2. User defined function

Library Functions

The library functions are pre defined function. A user cannot understand the internal working of the standard functions and cannot be modified but can only use all standard functions. The user must include the prototype declarations (header files) to use the standard library functions. The example of standard functions are scanf(), printf(), sqrt(), abs(), log(), sin(), pow() etc.

User defined functions

User defined functions are the functions defined by the user according to the requirements. The user understands the internal working of the function. The **main()** functions is also a user defined function except that the name of the function, the number of arguments, and the argument types are defined by the language.

You can write as many functions as you like in a program as long as there is only one main (). As long as these functions are saved in the same file, you do not need to include a header file. They will be found automatically by the compiler.

The general form of function

```
Return_type function_name ( parameters_list)
{
  Local variable declaration;              /* within the function */
   Statements1;
   ……..
   ……..
   Statement n;
  return (expresion);
}
```

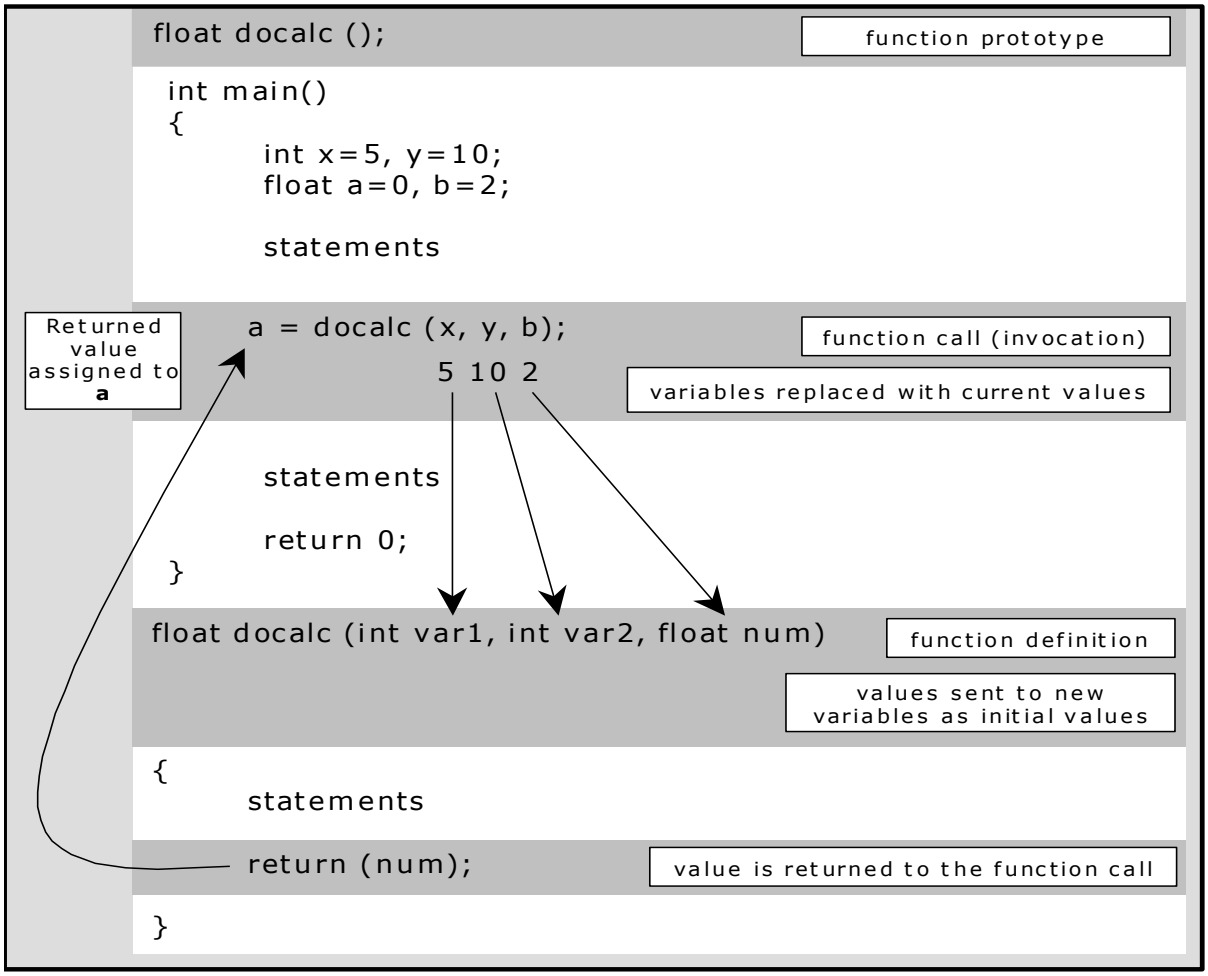
- ***Return_type*** specifies the type of value that the function's return statement returns. If nothing is returned to the calling function, then data type is void.
- ***function_name*** is a user-defined function name. It must be a valid C identifier.
- ***Parameter list*** declares the variables that will receive the data sent by the calling function. They serve as input data to the function to carry out the specified task. Since they represent actual input values, they are often referred to as ***formal parameters***.

- **return** is a keyword used to send the output of the function, back to the calling function. It is a means of communication from the called function to the calling function. There may be one or more return statements. When a return is encountered, the control is transferred to the calling function.

- All the statements placed between the left brace and the corresponding right brace constitute the ***body of a function***.
  { is the beginning of the function.

  } is the end of function.

The following figure shows the flow through a function:



Function Declaration (or Function Prototype)

Like variable, all functions in a C program must be declared, before they are invoked. Function declaration is also called as function prototype. It is declared in the declaration part of the main program. The default return value from a function is always an integer. If the function is returning a value other than an integer, it must be declared with the data type of the value it returns.

The general form of function prototype is as follows:

Return_type function_name(parameter_list);

This is very similar to the function header line except the terminating semicolon.

For example: long int factorial(int n);

Function Call

A function can be called by simply using the function name followed by a list of actual parameter (or arguments), if any, enclosed in parentheses.

The general form of a function call is as follows:

function_name (*actual parameters*);

Function Parameters (or Arguments)

Function parameters are the means of communication between the calling and called functions. They can be classified into *Actual parameters* and *Formal parameters*.

The ***Actual parameters***, often known as arguments, are specified in the function call. These are written within the parentheses followed by the name of the function. These are accepted in the main program (or calling function).

The ***Formal parameters*** (commonly called parameters) are the parameters given in the function declaration and function definition. These are not the accepted values but they receive values from the calling function. Parameters must be written within the parentheses followed by the name of the function, in the function definition.

*The **return** statement*

The user defined function uses *return* statement to return the value to the calling function. Exit from the called function to the calling function is done by the use of return statement. When *return* statement is executed it always return1.

1. ***return 0*** this statement return zero to the operating system if the value entered by the user is 1 or negative.
2. ***return NULL*** the statement return NULL.
3. ***return(expression)*** the statement return the value
   Example: ***return(a+b+c);***

return(&a);

return(*c);

return(sqrt(r));

Consider the following program. In this program, in main( ) we receive the values of a, b and c through the keyboard and then output the sum of a, b and c. However, the calculation of sum is done in a different function called calsum( ). If sum is to be calculated in calsum( ) and values of a, b and c are received in main( ), then we must pass on these values to calsum( ), and once calsum( ) calculates the sum we must return it from calsum( ) back to main( ).

Example program

```
/* Sending and receiving values between functions */
calsum(int, int, int);
main( )
{
 int  a, b, c, sum ;
 printf ( "\nEnter any three numbers " ) ;
 scanf ( "%d %d %d", &a, &b, &c ) ;
 sum = calsum ( a, b, c ) ;
 printf ( "\nSum = %d", sum ) ;
}

calsum ( int x, int y, int z )
{
 int  d ;
 d = x + y + z ;
 return ( d ) ;
}
```

Output

```
Enter any three numbers 10 20 30
```

Sum = 60

2. Types of user defined function (categories of function)

Depending upon the arguments present, return value sends the result back to the calling function. Based on this, the functions are divided into four types.

1. Without argument and without return value
2. With argument but without return value
3. With argument and with return value
4. Without argument and but with return value

Without argument and without return value

Calling function	analysis	Called function

main()
{
……
……
abc();
…….
…….
}
	→No Arguments are passed→
	
	←No Values are sent back←
		abc()
		{
		…….
		…….
		…….
		
		}

1. Neither the data is passed through the calling function nor data is sent back from the called function.
2. There is no data transfer between calling and the called function.
3. The function is only executed and nothing is obtained.
4. If such functions are used to perform any option, they act independently. They read data values and print result in the same block.
5. Such functions may be useful to print some messages, draw a line or split the line etc.

Example program

The following program illustrates the function with no arguments and no return value.

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

```
{
void sum();
    clrscr();
    Sum();
}

void sum()                                  /*no return value */
{
    Int a=10,b=20,c;
    C = a + b;

    printf ("\nThe sum of Two numbers is :%d" ,c)
}
```

Output

```
The sum of Two numbers is : 30
```

With argument but without return value

Calling function	analysis	Called function

main()
{
……
……
abc(x);
…….
…….
}
	→ Arguments are passed→
	
	←No Values are sent back←
		abc(y)
		{
		…….
		…….
		…….
		
		}

1. In the above functions are passed through the calling function. The called function operates on the values. But no result sent back.

2. Such functions are partly dependent on the calling function. The result obtained is utilized by the called function and there is no gain to the main()

Example program

The following program illustrates the function with arguments and no return value.

```
# include <stdio.h>

main ()
{
int a=10,b=20;
void sum(int a, int b);
     clrscr();
     Sum(a,b);
     getch();
}

void sum(int a , int b)                              /*no return value */
{
     int c;
     c = a + b;
     printf ("\nThe sum of Two numbers is :%d" ,c)
}
```

Output

```
The sum of Two numbers is : 30
```

With arguments and return value

Calling function	Analysis	Called function
main() {	→Arguments are passed→	abc(y) {

int z;
……
Z=abc(x);
…….
…….
}
	←Values are sent back←	…….
		y++;
		…….
		…….
		return(y);
		}

In the above example the copy of actual argument is passed to the formal argument i.e., value of 'x' is assigned to 'y'. The return statement returns the increased value of 'y'. The returned value is collected by 'z'. Here data is transferred between calling and the called functions i.e., communication between function is made.

Example program

Write a C program to send values to user-defined function and receive and display the return value.

```
#include<stdio.h>
int sum(int,int,int);
main()
{
  int a,b,c,s;
  printf("Enter three numbes:");
  scanf("%d%d%d",&a,&b,&c);
  s=sum(a,b,c);
  printf("Sum=%d",s);
 }
int sum(int x,int y,int z)
{
  return(x+y+z);
}
```

Output:

```
Enter three numbers:
7 5 4
Sum=16
```

With arguments and but return values

Calling function	Analysis	Called function

main()
{
int z;
……
Z=abc();
…….
…….
}
	→No Arguments are passed→
	
	←Values are sent back←
		abc()
		{
		int y=5;
		…….
		…….
		…….
		return(y);
		}

In the above types of function no argument(s) are passed through the main() function. But the called function returns the values. The called function is independent. It reads values from the keyboard or generates from initialization and return the value. Here both the calling and the called functions are partly communicated with each other.

Example program

Write a c program to receive values from the user-defined function without passing any value through main().

```
#include<stdio.h>
main()
{
 int sum(), s;
 s=sum();
 printf("Sum=%d",s);
}
int sum()
{
 int x,y,z;
 printf("Enter three values:");
scanf("%d%d%d",&x,&y,&z);
return(x+y+z);
}
```

Output:

```
Enter three values:
3 5 4
Sum=13
```

3. Understanding the scope of a function

There are two kinds of variables

1. Local variable
2. Global variable

Local variable

The local variables are defined within the body of the function or the block. This variable can access only within the function or block. Other functions cannot access these variables.

Example program:

Write a program to show how similar variable names can be used in functions.

```
main()
{
  int b=10,c=5;
  clrscr();
   printf("\n in main( ) b=%d c=%d",b,c);
   fun( );
}
fun( )
{
 int b=20,c=10;
 printf("\n in fun( ) b=%d c=%d",b,c);
}
```

Output:

```
in main( ) b=10 c=5
in fun( ) b=20 c=10
```

Global variable

Global variables are defined outside the main( ) function. Multiple functions can use them.

Example program:

Write a program to show the effect of global variables on different function.

```
int b=10,c=5;
main()
{
  clrscr();
   printf("\n in main( ) b=%d c=%d",b,c);
   fun( );
   b++;
   c--;
  printf("\n again in main( ) b=%d c=%d",b,c);
 }
 fun( )
{
  b++;
  c--;
  printf("\n in fun( ) b=%d c=%d",b,c);
}
```

Output:

```
in main( ) b=10 c=5
in fun( ) b=11 c=4
again in main( ) b=12 c=3
```

4. Scope rules

The scope rules of a language are the rules that govern whether a piece of code knows about or has access to another piece of code or data. Scope is the region of a program in which a variable is a available for use. Lifetime of a variable is the duration of time in which a variable exists in the memory during execution.

Rules of use

1. The scope of a global variable is the entire program file.

2. The scope of a local variable begins at point of declaration and ends at the end of the block or function in which it is declared.

3. The scope of a formula function argument is its own function

4. The lifetime (or longevity) of an auto variable declared in main is the entire program execution time, although its scope is only the main function.

5. The life of an auto variable declared in a function ends when the function is exited.

6. A static local variable, although its scope is limited to its function, its lifetime extends till the end of program execution.

7. All variables have visibility in their scope, provided they are not declared again.

5. Type Qualifiers

C provides three type Qualifiers const, volatile, restrict. Const and volatile qualifiers can be applied to any variable. But restrict qualifiers may only applied to pointer.

Constant variable

A variable value cannot change during program execution by declaring the variable as constant. The keyword const is placed before the declaration. For the const pointer, place the keyword between * and identifier.

Syntax:

const int a;

Here a is constant and its value cannot be changed.

int *const x;

in the above example, the pointer to x is constant. The value that x points can be changed, but the value of y cannot change.

const int *x; (or) int const *x;

in the above example, both the statements give same meaning. The value that x points to is a constant integer and cannot be changed. However, the value of x can be changed.

Volatile variable

Variables that can be changed at any time by external programs or the same program are called as volatile variables. The keyword volatile is placed before declaration.

Syntax:

volatile int x;

volatile const int y;

volatile int *z; (or) int *volatile z;

The variable x value can be changed by any program at any time and the variable y can be changed in the current program but not by the external programs. The variable z is a pointer to a volatile int.

Restrict variable

The restrict type qualifier may only be applied to a pointer. A pointer declaration that uses this type qualifier establishes a special association between the pointer and the object it accesses, making that pointer and expressions based on that pointer, the only ways to directly and indirectly access the value of that object.

Syntax

int * restrict x;

6. Storage classes

The storage class of a variable tells the compiler.

- The storage area of the variable.
- The initial value of variable if not initialized.
- The scope of the variable.
- Life of the variable i.e., how long the variable would be active in the program.

There are four types of storage class.

1. Automatic
2. External or global
3. Static
4. Register.

Automatic storage class:

When a variable is declared as a auto, it is stored in the memory. The default value of the variable will be garbage value. Scope of the variable is within the block where it is defined and the life of the variable is until the control remains within the block.

Syntax:

auto data_type var_list;

Example:

auto int a,b;

Example for auto variables

```
main()
{
int a=1000;
fun2();
printf("%d",a);
}
fun1()
{
int a=10;
printf("%d",a);
}
fun2()
{
int a=100;
fun1();
printf("%d",a);
}
```

Output

```
10
100
1000
```

External storage class

When a variable is declared as extern, it is stored in the memory. The default value is initialized to zero. The scope of the variable is global and life of the variable is until the program execution comes to an end. An extern variable is also called as global variable.

Syntax:

extern data_type var_list;

Example:

extern int a,b;

Example

```
int a=25;
main()
{ extern int a;
printf("a=%d",a);
printf("a=%d",fun1());
```

```
}
```

Output

```
a=25
a=35
```

Static storage class

When a variable is declared as static, it is stored in the memory. The default value of the variable will be zero. scope of the variable is within the block where it is defined and the life of the variable persists between different function calls.

Syntax:

static data_type var_list;

Example:

static int a,b;

Example

```
main()
{
int c;
for(c=1;c<=3;c++)
fun();
}
fun()
{
static int a=5;
a=a+3;
printf("a=%d\n",a);
}
```

Output

```
a=8
a=11
a=14
```

Register storage class

When a variable is declared as register, it is stored in the CPU registers. The default value of the variable will be garbage value. Scope of the variable is within the block where it is defined and the life of the variable is until the control remains within the block.

Syntax:

register data_type var_list

Example:

register int x,y;

Example

```
#include<stdio.h>
main()
{   register int j=1;
  do
  {  printf("%d",j);

 j++;
  }while(j<=10);
}
```

Output

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

7. Function Arguments (Call by value and call by reference)

If a function is to accept arguments, it must declare the parameter s that will receive the values of the arguments.

There are two ways in which we can pass arguments to the function.

Call by value

In this type value of actual arguments are passed to the formal arguments and the operation is done on the formal arguments. Any change made in the formal argument does not affect the actual arguments because formal arguments are photo copy of actual arguments. Changes made in the formal arguments are local to the block of the called function.

Example program

```
#include<stdio.h>
main()
{
 int change(int,int);
 int x,y;
 printf("Enter values of x&y:");
 scanf("%d%d",&x,&y);
 printf("x=%d y=%d",x,y);
```

```
 change(x,y);
 printf("\n x=%d  y=%d",x,y);
}
int change(int a, int b)
{
  int k;
  k=a;
  a=b;
  b=k;
}
```

Output:

```
Enter values of x&y: 5 4
x=5 y=4
x=5 y=4
```

Call by reference

In this type instead of passing values, addresses (reference) are passed. Function operates on addresses rather than values. Here, the formal arguments are pointers o the actual arguments. In this type formal arguments are point to the actual argument. Hence changes made in the arguments are permanent.

Example program

```
 #include<stdio.h>
main()
{
  int change(int*,int*);
  int x,y;
 printf("Enter values of x&y:");
 scanf("%d%d",&x,&y);
 printf("\n x=%d y=%d",x,y);
 change(&x,&y);
 printf("\n x=%d  y=%d",x,y);
}
int change(int *a, int *b)
```

```
{
  int k;
 k=*a;
 *a=*b;
 *b=k;
 printf("x=%d y=%d",*a,*b);
}
```

Output:

```
Enter values of x&y: 5 4
x=5 y=4
x=4 y=5
x=4 y=5
```

Pointers

1. Pointer definition

Pointer is a variable which stores the address of another variable. The pointer variable declared same as a normal variable declaration but the difference is the pointer variable is followed by a asterisk (*) symbol which is also called as indirection operation.

Pointer declaration

A pointer declaration consists of a base type, an *, and the variable name.

Syntax:

Datatype *pointer_variable name;

Where,

Datatype→it is basic datatype of the pointer

Pointer_variable name→ name of the pointer

Technically, any type of pointer can point anywhere in memory.

Example:

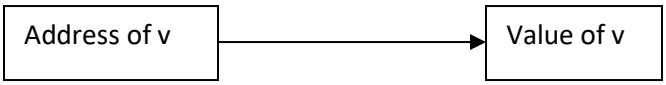
```
int *pv;
int v=10;
```

Now let us assign the address of v to another variable, `pv`. Thus,

```
pv = &v
```

This new variable is called a *pointer* to `v`, since it "points" to the location where `v` is stored in memory. Remember, however, that `pv` represents `v's` *address*, not its value. Thus, `pv` is referred to as a *pointer variable*.

The relationship between pv and v illustrated below



pv v

Relationship between pv and v (where pv=&v and v=*pv)

Example programs

Write a program to display the value of variable and its location – using pointer.

```
#include<stdio.h>
#include<conio.h>
main()
{
 int v=10,*p;
 clrscr();
 p=&v;
 printf("\n Address of v=%u",p);
 printf("\n Value of v=%d",*p);
 printf("\n Address of p=%u",&p);
}
```

Output

```
Address of v=4060
Value of v=10
Address of p=4062
```

2.The pointer operators

There are two pointer operators: * and &. The & (address) is a unary operator that return the memory address of the operand. The * (Pointer operator) is complement of &.it is a unary operator that returns the value located at the address. Both are prefix unary operators.

Example:

m=&count; (address)
q=*m; (pointer operator)

3. Pointer expressions

In general, expressions involving pointer conform to the same rules as other expressions. Pointer expressions, such as assignments, conversions, and arithmetic.

Pointer assignments:

You can use a pointer on the right hand side of an assignment statement to assign its value to another pointer. When both pointers are the same type, the situation is straight forward.

Example:

```
int x=99;
int *p1,*p2;
p1=&x;
p2=p1;
```

Example program

```
#include<stdio.h>
main()
{
 int x=99;
 int *p1,*p2;
 p1=&x;
 p2=p1;
 printf("values at p1 &p2:%d %d\n",*p1,*p2);
 printf("address pointed to by p1 and p2:%u %u",p1,p2);
}
```

Output

```
values at p1 &p2:99 99
address pointed to by p1 and p2:65490 65490
```

Pointer conversion

- Suppose we have to declare integer pointer, character pointer and float pointer then we need to declare 3 pointer variables.
- Instead of declaring different types of pointer variable it is feasible to declare single pointer variable which can act as integer pointer, character pointer and float pointer.
- void* is called a generic pointer. Void pointer cannot be dereferencing without explicit type conversion.

Void pointer basic:

1. In C **General Purpose Pointer** is called as void Pointer.
2. It does not have any data type associated with it
3. It can store address of any type of variable
4. A void pointer is a C convention for a raw address.
5. The compiler has no idea what type of object a void Pointer really points to ?

Declaration of void pointer

```
void *pointer_name;
```

Example program

```
#include<stdio.h>
int x;
float y;
char z;

void *p;
main()
{
 clrscr();
 p=&x;
 *(int*)p=5;
 printf("x=%d\n",x);
 p=&y;
 *(float*)p=5.4;
 printf("y=%f\n",y);
 p=&z;
 *(char*)p='s';
 printf("z=%c\n",z);
}
```

Output

```
x=5
y=5.400000
z=s
```

Pointer arithmetic

Arithmetic operations on pointer variables are also possible. Increase, decrease, prefix and postfix operations can be performed with the help of the pointer. The effect of these operations are shown in the below.

Data type	Initial address	Operation		Address after operation		Required bytes
int i=2	4046	++	--	4048	4044	2
char c='x'	4053	++	--	4054	4053	1
float f=2.2	4058	++	--	4062	4054	4
long l=2	4060	++	--	4064	4056	4

From the above table we can observe that on increase of the pointer variable for integers the address is increased by two i.e., 4046 is original address and on increase its value will be 4048 because integer require two bytes. Similarly, for character, floating point number and long integers requires 1, 4, 4 bytes respectively.

Example

```
#include<stdio.h>
#include<conio.h>
main()
{
    int i, *p;
    printf("enter value of i=");
    scanf("%d",&i);
    p=&i;
    clrscr();
    printf("Address of i=%u\n",p); printf("Address of i=%u\n",++p);
    printf("Address of i=%u\n",p++); printf("Address of i=%u\n",--p);
    printf("Address of i=%u\n",p--); printf("Address of i=%u\n",p);
```

```
}
```

Output

```
Enter value of i=8
Address of i =4060
Address of i =4062
Address of i =4062
Address of i =4062
Address of i =4062
Address of i =4060
```

Pointer comparisons

You can compare two pointers in a relational expression.

Example2

```
#include<stdio.h>
#include<conio.h>
main()
{
    int a=25, b=10,*p,*q;
    p=&a;
    q=&b;
    clrscr();
    if(*p>*q)
    printf("a is largest");
    else
    printf("b is largest");
```

```
}
```

Output

```
a is largest
```

4. Pointers and arrays

There is a close relationship between pointers and arrays. Array name by itself is an address or pointer. It points to the address of the first element (0th element of an array). The element of the array together with their addresses can be displayed by using array name itself. Array elements are stored in contiguous memory locations.

Recall that an array name is really a pointer to the first element in the array. Therefore, if x is a one-dimensional array, then the address of the first array element can be expressed as either &x [0] or simply as x. Moreover, the address of the second array element can be written as either &x [1] or as (x + 1), and so on.

Example

```
# include<stdio.h>
main( )
{
int x[l0] = {10, 1 1 , 12, 13, 14};
int i ;
for ( i = 0; i <5; i++)
 {
/ * display an array element */
printf ( " \ n i = %d x [i] = %d  *( x + i ) = %d", i, x [i] , *(x + i)) ;
/* display the corresponding array address * /
printf ( " &x[i]= %X x+i= %X", & x [ i ] , ( x + i ) ) ;
}
}
```

Output

```
i=0     x [ i ] = 10     * ( x + i ) = 10     &x[i]= 72     x+i= 72
i=1     x [ i ] = 11     * ( x + i ) = 11     &x[i]= 74     x+i= 74
i=2     x [ i ] = 12     * ( x + i ) = 12     &x[i]= 76     x+i= 76
i=3     x [ i ] = 13     * ( x + i ) = 13     &x[i]= 78     x+i= 78
i=4     x [ i ] = 14     * ( x + i ) = 14     &x[i]= 79     x+i= 79
```

Arrays of pointers

Pointer can be arrayed like any other data type. it is nothing but collection of addresses. Here, we store address of variables for which we have to declare an array as a pointer.

Example:

int *p[10];

To assign the address of an integer variable called **var** to the third element of the pointer array, write

P[2]=&var;

Example program

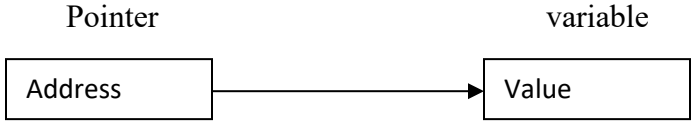
```
#include<stdio.h>
#include<conio.h>
main()
{
int *p[3];
int a[3]={1,2,3},i;
clrscr();
for(i=0;i<3;i++)
p[i]=&a[i];
printf("the elements are:");
for(i=0;i<3;i++)
printf("%d",*p[i]);
}
```

Output

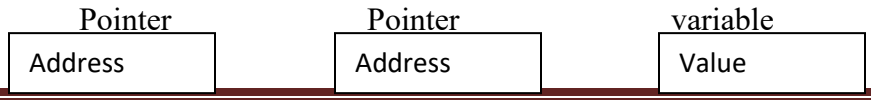
```
The elements are: 1 2 3
```

5. Pointers to pointers (or) multiple indirection

Pointer is known as a variable containing address of another variable. The pointer variable containing address of another pointer variable is called as pointer to pointer. This is chain continued to any extent.



Single indirection



Multiple indirection

Example program:

```
#include<stdio.h>
main()
{
 int x,*p,**q;
x=10;
p=&x;
q=&p;
printf("%d",**q);
}
```

Output

```
10
```

6. Initializing pointers

A pointer that does not currently point to a valid memory location is given the value NULL (which is zero). Null is used because C guarantees that no object will exist at the null address. Thus, any pointer that is null implies that it points to nothing and should not be used.

One way to give a pointer a null value is to assign zero to it. for Example,

char *p=0;

Additionally, many of C's headers, such as <stdio.h> , define macro NULL, which is an null pointer constant. Therefore, you will often see a pointer assigned null using a pointer statement such as this:

p=NULL;

for example, the following sequence, although in correct, will still be compiled without error:

int *p=0;

p=10 / wrong!*/

7. Pointers to functions

A function has a physical location in memory that can be assigned to a pointer. This address is entry point of the function and it is the address used when the function is called. Once a pointer points to a function, the function can be called through that pointer. Function pointers also allow functions to be passed as arguments to other functions.

Example

Write a program to exchange two numbers using pointers.

```
#include<stdio.h>
main()
{
 int change(int*,int*);
 int x,y;
 printf("Enter values of x&y:");
 scanf("%d%d",&x,&y);
 printf("\n main (before) function x=%d y=%d",x,y);
 change(&x,&y);
 printf("\n main (after) function x=%d  y=%d",x,y);
}

int change(int *a, int *b)
{
  int k;
  printf("\n sub(before) function x=%d y=%d",*a,*b);
  k=*a;
  *a=*b;
  *b=k;
  printf("\n sub(after) function x=%d y=%d",*a,*b);
}
```

Output

```
Enter values of x&y: 5 4
Main (before) function x=5 y=4
sub (before) function x=5 y=4
sub (after) function x=4 y=5
main (after) function x=4 y=5
```

8. C's dynamic allocation functions

C language requires the number of elements in an array to be specified at compile time. But we may not be able to do so always. Our initial judgment of size, if it is wrong, may cause failure of the program or wastage of memory space.

Many languages permit a programmer to specify an array's size at run time. Such languages have the ability to calculate and assign, during execution, the memory space required by the variables in program. The process of allocating memory at run time is known as dynamic memory allocation.

1) malloc( ) The name malloc stands for "memory allocation". A block of memory may be allocated using the function malloc. The malloc function reserve a block of memory of specified size .This means that we can assign it to any type of pointer. It takes the following form.

```
Ptr=(cast-type*) malloc (byte-size);
```

Ptr is a pointer of type cast-type. The malloc returns a pointer (of cast type) to an area of memory with size bytes - size.

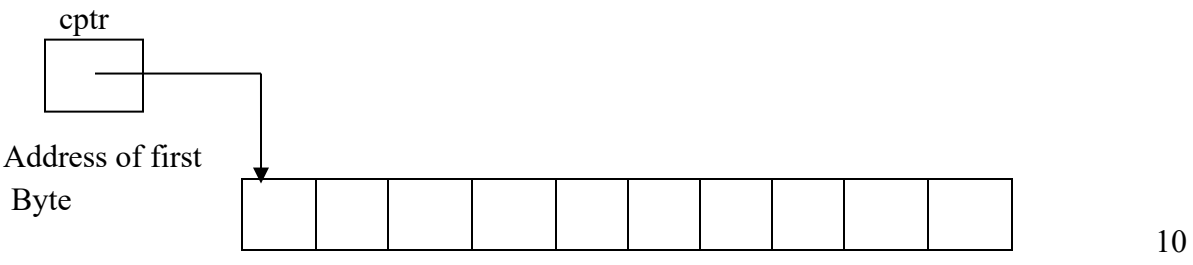
Example

X=(int*)malloc(100*size(int));

On successful execution of this statement, a memory space equivalent to "100 times the size of an int" byte is reserved and the address of the first byte of the memory allocated is assigned to the pointer x of type of int.

Similarly, the statement

cptr=(char*) malloc (10)



Note that the storage space allocated dynamically has no name and therefore its content can be accessed only through a pointer.

Example

```
#include<stdio.h>
#include<conio.h>
int main()
{
 int *a,n,i;
 clrscr();
 printf("enter the size of the array\n");
 scanf("%d",&n);
a=(int*)malloc(n*sizeof(int));/*allocating memory*/
printf("enter elements:")
for(i=0;i<n;i++)
scanf("%d",a+i);/*reading data*/
printf("elements are:");
for(i=0;i<n;i++)
 printf("%d ",*(a+i));/* displaying the data*/
 free(a);
return 0;
}
```

Output

```
Enter the size of the array 5
Enter elements: 10 11 12 13 14
Elements are:10 11 12 13 14
```

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

Syntax:

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

example: ptr=(float*)calloc(25,sizeof(float));

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

3. realloc(): this function adjusts the allocated memory to size and copies old data if required. If success this function returns new data blocks address otherwise it returns NULL.

Syntax:

ptr=realloc(ptr,newsize);

4. Free(): this function requires address of the memory block which is allocated through either malloc, calloc, realloc to de-allocate memory.

free(ptr);

This statement causes the space in memory pointer by ptr to be de-allocated.

9. Problems with pointers

When a pointer is used incorrectly, or contains the wrong value, it can be a very difficult to bug to find. To help you avoid them, a few of the more common errors are discussed here.

Example1:

```
int x=10,*p;
*p=x; /*error, p not initialized*/
```

Example2:

```
int x=10,*p;
p=x; /*here, assign the value, but assign the address */
```

Example3:

```
        char s[50],y[80];
char *p1,*p2;
p1=s;
p2=y;


if(p1<p2)…
```

Is generally an invalid concept.(in very unusual situations, you might use something like this to determine the relative position of the variables. but this would be rare.

10. 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

```
int *p;
int sample[50];
p=sample;
```

You can 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.

11. Passing single dimensional array to function

In C, you cannot pass an entire array as an argument to a function. However pass a pointer to an array by specifying the array's name without an index. for example

```
int main( )
{
  int i[10];
   func1(i);
 /*…..*/
}

Void func1(int *x)
{
        /*…….*/

}
```

1. Command line arguments

An executable program that performs a specific task for operating system is called as command. The commands are issued from the prompt of operating system. some arguments are to be associated with the commands hence these arguments are called as command line arguments. These associated arguments are passed to the program.

In c language every program starts with a main() function and that it marks the beginning of the program. We have not provided any arguments so far in the main() function. Here, we can make arguments in the main like other functions.

The main() function can receive two arguments and they are

1. argc
2. argv

argc an argument argc counts total number of arguments are passed from command prompt. It returns a value which is equal to total number of arguments passed through the main()

argv is a pointer to an array of character strings which contains names of arguments. Each word is an argument.

Example

Write a program to display number of arguments and their names

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

main(int argc, char *argv[ ] )
{
  int x;
  clrscr();
  printf("\n Total number of arguments are %d\n",argc);
  for(x=0;x<argc;x++)
  printf("%s\t",argv[x]);
  getch();
  return 0;
}
```

Output

Total numbers of arguments are 4

C:\tc\c.exe A B C

Explanation

To execute this program one should create its executable file and run it from the command prompt with required arguments. The above program is executed using following steps

a) Compile the program

b) Make its exe file (executable file)
c) Switch to the command prompt. (c:\tc)
d) Make sure that exe file is available in the current directory
e) Type following bold line

c:\Tc> c.exe HELP ME

in the above example c.exe is an executable file and "HELP ME" are taken as arguments. The total number of arguments including the program file name is 3.

2. Declaring variable length parameter lists

You can specify a function that has a variable number of parameters. For example, this prototype specifies that func( )will have at least two integer parameters and unknown number (including 0) of parameters after that:

int func(int a, int b,….)

This form of declaration is also used by a functions definition. Any function that uses a variable number of parameters must have at least one actual parameters.

For example, this is incorrect:

int func(…); /* illegal */

Structures

1. Structure definition

A structure is a collection of one or more variables of different data types, grouped together under a single name. a structure allows the programmer to create and manipulate a set of different types of data items. In C, all the data items can be grouped together in one structure using **struct**.

Declaration and Initialization of Structures

Each and every structure must be defined and declared before it appears in a C program. The general form of structure declaration is as follows.

```
struct   struct _type
{
 datatype  variable1;
 datatype variable2;
```

struct→ structure declaration always starts with **struct** keyword.

struct_type→it is the name of the structure. it is derived datatype.

datatype→ basic data types (like int, float, char etc.).
variable1, variable2……. variablen →are different data items or structure variables. And each is called a member of a structure. These are also called the fields.

The left brace indicates the beginning of a structure and the right brace specifies the end of the structure. The body of the structure is terminated by semicolon.

After defining structure we can create variable as given below.

```
struct struct_type v1,v2,v3;
```

Here, v1,v2 and v3 are variables of structure struct_type. this is similar to declaring variables of any data type.

int v1,v2,v3;

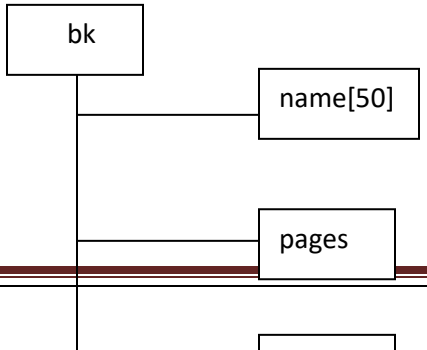
The declaration defines the structure but this process doesn't allocate memory. The memory allocation takes places only when variable are declared.

Example of the structure declaration

```
struct  book
{
char name[50];
int pages;
float price;
};
struct  book bk;
```

or

```
struct  book
{
char name[50];
int pages;
float price;
}bk;
```



In The above example a structure of book is created. It consists of three numbers name[50] of char data type, pages of int data type an price of float data type.

```
struct book bk;
```

The above line creates variable bk of type book and reserves total 56 bytes. (50 bytes for name[50], 2 bytes for pages and 4 bytes for price).

Initialize structure elements. The general form of structure initialization is as follows.

```
struct struct_type v1={e1,e2…..en};
```

el, e2…..en→ it is the number of elements.
In order to initialize structure elements with certain values following statement is used.

```
struct book bk={"c and ds",340,325.32};
```

Accessing structure members

Processing of a structure is mainly concerned with accessing a structure member. Each member of a structure is accessed with the dot operator. That is, the decimal point. To access a particular member, the dot operator must be placed between the name of the structure and the name of the structure member.

All the members of structure are related to variable bk. Through all the three members of structure can be accessed.

```
structure_variable.member        or        bk.name
```

The period (.) sign is used to access the structure member.

We can directly assign values to members as given below

```
bk.name="c and ds";
bk.pages=340;
bk.price=325.32;
```

Example program

Write a program to define a structure and initialization its member variables.

```
/* structure initilization and printing */
#include<stdio.h>
#include<conio.h>
main()
```

```
{
    struct book
    {
     char name[50];
     int pages;
    float price;
    };
    struct book bk={"c and ds",340,325.32};
     clrscr();
     printf("\n book name:%s",bk.name);
     printf("\n no.of pages:%d",bk.pages);
     printf("\n book price:%f",bk.price);
}
```

Output

```
book name:c and ds
 no.of pages:340
 book price:325.320007
```

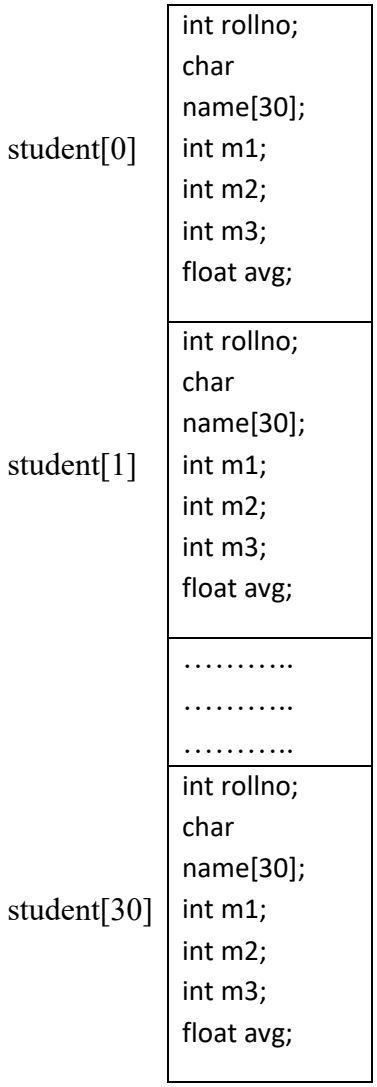
2. Array of Structures

Whenever the same structure is to be applied to a group of people, items or applications we will have to use an array of structures. In an array of structures each element is a structure itself.

For example, if the final internal assessment is to be made for a class of 30 students, then the teacher may make use of an array of 30 structures and each student structure may be defined as,

```
struct stu_rec
{    int rollno;
     char name[30];
     int m1;
     int m2;
     int m3;
     float avg;
} student[30];
```

Thus, an array of structures, would be initialized as



Example program

```
/* students marks details using array of structure */
#include<stdio.h>
#include<conio.h>
main()
{
struct stu_rec
{
  int rollno;
  char name[30];
  int m1;
  int m2;
  int m3;
  float avg;
};
struct stu_rec s[4];
int total,i;
clrscr();
for(i=1;i<=3;i++)
{
 printf("Enter %d student details\n",i);
 printf("Enter student rollno, name, m1, m2, m3:");
scanf("%d%s%d%d%d",&s[i].rollno,s[i].name,&s[i].m1,&s[i].m2,
&s[i].m3);
}
printf("----------------------------------------------------------\n");
printf("Rollno\tName\tmark1\tmark2\tmark3\tavg\n");
printf("----------------------------------------------------------\n");
for(i=1;i<=3;i++)
{
 total=s[i].m1+s[i].m2+s[i].m3;
 s[i].avg=total/3;

printf("%d\t%s\t%d\t%d\t%d\t%f\n",s[i].rollno,s[i].name,s[i].m1,s[i
].m2,s[i].m3,s[i].avg);
}
printf("\n----------------------------------------------------------");
}
```

Output

```
Enter 1 student details
Enter student rollno, name, m1, m2, m3:
401
naveen
30
40
50
Enter 2 student details
Enter student rollno, name, m1, m2, m3:
402
vijay
30
20
40
Enter 3 student details
Enter student rollno, name, m1, m2, m3:
403
ajay
20
40
30
-----------------------------------------------------
Rollno  Name    mark1   mark2   mark3   avg
-----------------------------------------------------
401     naveen    30      40      50    40.000000
402     vijay     30      20      40    30.000000
403     ajay      20      40      30    30.000000
-----------------------------------------------------
```

Array within structures

C permits the use of arrays as structure members. We have already used arrays of characters inside a structure. Similarly, we can use single or multi-dimensional arrays of type int or float. For example, the following structure declaration is valid.

```
struct marks
{
  int number;
```

```
  float subject[3];
} student [2];
```

Here, the member subject contains three elements, subject[0], subject[1] and subject[2]. These elements can be accessed using appropriate subscripts.

3. Passing Structures to Functions

There are three methods by which the values of a structure can be transferred from one function to another.

The first method is to pass each member of the structure as an actual argument of the function call. The actual arguments are then treated independently like ordinary variables. This is the most elementary method and becomes unmanageable and inefficient when the structure size is large.

The second method involves passing of a copy of the entire structure to the called function. Since the function is working on a copy of the structure, any changes to structure members within the function are not reflected in the original structure (in the calling function). It is, therefore, necessary for the function to return the entire structure back to the calling function. All compilers may not support this method of passing the entire structure as a parameter.

The third approach employs a concept called pointers to pass the structure as an argument. In this case, the address location of the structure is passed to the called function. The function can access indirectly the entire structure and work on it. This is similar to the way arrays are passed to functions. This method is more efficient as compared to the second one.

Example program

Write a program to pass address of structure variable to user defined function and display the contents.

```
/* passing address of structure variable */
#include<stdio.h>
#include<conio.h>
struct book
{
 char name[35];
 char author[35];
 int pages;
```

```
};
main()
{
 struct book bk1={"c&ds","murahari",320};
 clrscr();
 display(&bk1);
}
display(struct book *bk2)
{
 printf("\n%s by %s of %d pages",bk2->name,bk2->author,bk2->pages);
}
```

Output

```
c&ds by murahari of 320 pages
```

Explanation

In the above program structure **book** is defined before **main( )**. In the **main( )** function b1 is an object declared and initialized. The address of object **bk1** is passed to function **display( )**. In the function **display( )**. In the function **display(**) the address is assigned to pointer *bk2 that is a pointer to the structure book. Thus, using**->**operator contents of structure elements are displayed.

Example program

Write a program to pass structure elements to function display( ) and print the elements.

```
/* passing structure elements to funtion */
#include<stdio.h>
#include<conio.h>
struct book
{
 char name[35];
 char author[35];
 int pages;
};
main()
{
```

```
 struct book bk1={"c&ds","murahari",320};
 clrscr();
 display(bk1.name,bk1.author,bk1.pages);
}
display(char *s,char *t,int p)
{
 printf("\n%s by %s of %d pages",s,t,p);
}
```

Output

```
c&ds by murahari of 320 pages
```

Explanation

In the above example structure name has member variable like a character array name[35],array author[35]and pages of integer type. we have passed the base address of name but values of pages. Thus, here values are passed using call by reference and call by value methods. Instead of passing each element also one can pass entire structure into the function.

Example program

Write a program to pass entire structure to user defined function.

```
/* passing entire structure to function */
#include<stdio.h>
#include<conio.h>
struct book
{
 char name[35];
 char author[35];
 int pages;
};
main()
{
 struct book bk1={"c&ds","murahari",320};
 clrscr();
 display(bk1);
}
display(struct book bk2)
```

```
{
 printf("\n%s by %s of %d pages",bk2.name,bk2.author,bk2.pages);
}
```

Output

```
c&ds by murahari of 320 pages
```

Explanation

In the above program structure book is defined outside the **main ( )**. So it is global and every function can access it. The object defined on **struct book bk1** is passed to function **print( )**. The formal argument (object) of function **print( )** receives contents of object bk1. Thus, using **dot** operator contents of individual elements are displayed.

4. Structure pointers

C allows pointers to structures just as it allows pointers to any other type of object. Here, starting address of the member variables can be accessed. Thus, such pointers are called structure pointers.

Example

```
struct book
{
  char name[25];
  int pages;
 float price;
};
struct book *bk;
```

in the above example *bk is pointer to structure book. The syntax for using pointer with member is as given below.

1. bk ->name
2. bk ->pages
3. bk ->price

By executing these three statements starting address of each member can be estimated.

Example program

Write a program to declare pointer to structure and display the contents of the structure.

```
/* size of the structure elements */
```

```
#include<stdio.h>
#include<conio.h>
main()
{
 struct book
 {
  char name[50];
  int pages;
  float price;
 };
 struct book bk={"c&ds",340,325.32};
 struct book *ptr;
 ptr=&bk;
 clrscr();
 printf("%s %d %f\n",bk.name,bk.pages,bk.price);
 printf("%s %d %f",ptr->name,ptr->pages,ptr->price);
}
```

Output

```
c&ds 340 325.320007
c&ds 340 325.320007
```

5. Array and Structure within Structure (embedded structure)

A structure within structure is called an embedded structure. This declaration may take one of the two forms mentioned below.

1. Structure may completely be defined within the other structure.
2. There may be separate structures. The embedded structure is declared first and the outer structure is declared next.

We can also take object of one structure as member in another structure. Thus, structure within structure can be used to create complex data application.

For example,

```
struct employee
{
  int empno;
  char empname[50];
  float empsal;
```

```
  struct date dob;
};
```

In this declaration, struct date appears as an embedded structure. Therefore, it must be declared before struct employee. The structure date has the following definition.

```
Struct date
{
  int day;
  int month;
  int year;
};
```

Or, the above structure definition can be included within structure definition can be included within struct employee as follows:

```
struct employee
{
  int empno;
  char empname[50];
  float empsal;
 struct date
 {
   int day;
   int month;
   int year;
 };
  struct date dob;
};
struct employee e1;
```

Example program

Write a program for employee details structure within structure

```
/* Employee details structure within structure*/
#include<stdio.h>
#include<conio.h>
main()
{ struct employee
  {   int empno;
```

```
    char empname[50];
    float empsal;
   struct date
  {  int day;
     int month;
     int year;
   } dob;
 }; struct employee e1;
clrscr();
/* reading a structure within structure */
printf("Enter empno,empname,empsal &empdob:");
scanf("%d%s%f",&e1.empno,e1.empname,&e1.empsal);
scanf("%d%d%d",&e1.dob.day,&e1.dob.month,&e1.dob.year);
/* print a structure within structure */
printf("Employee details:");
printf("\nemployee number:%d",e1.empno);
printf("\nemployee name:%s",e1.empname);
printf("\nemployee salary:%f",e1.empsal);
printf("\ndate of birth:%d/%d/%d",e1.dob.day,e1.dob.month,e1.dob.year);
}
```

Output

```
Enter empno, empname, empsal &empdob:1
harshavardhan
30200.32
2
4
2010
Employee details:
Employee number: 1
Employee name: harshavardhan
Employee salary: 30200.320312
Date of birth: 2/4/2010
```

6. Union

Union is a variable, which is similar to the structure. It contains number of members like structure but it holds only one object at a time. In the structure each members has its own

memory location where as, members of unions have same memory locations. It can accommodate one member at a time in single area of storage. Union also contains members of types int, float, long, array, pointer etc. it allocate fixed specific bytes of memory for access of data types irrespective of any data type.

The union requires bytes that are equal to the number of bytes required for the largest members. For example, the union contains char, integer &long integer then the number of bytes reserved in the memory for the union is 4 bytes.

```
union   union _type
{
 datatype  variable1;
 datatype variable2;
  ……..
  ……..
 datatype  variablen;
};
```

Union→ union is keyword.

union_type→is a valid C identifier to denote the union.

datatype→ basic data types (like int, float, char etc.).

variable1, variable2……. variablen →are different data items or union variables. And each is called a member of a union. These are also called the fields.

Example (1) :

```
union number
{
 int n1;
 float n2;
};
```

Here, the number is a union with two members. The first member n1, is of type int and the second member n2, is of type float.

Declaration

Like structures declarations unions are declared before they are used in a C program. They can be declared in two styles.

1. within the union definition
2. outside the union definition

In the first style, the variables of type union are included between the closing brace and the semicolon. If there is more than one variable, then they must be separated by commas.

Example:

```
union number
```

```
{
 int n1;
 float n2;
}x;
```

Here, X is declared as a variable of type union number.

In the second style, the variables are declared on a separate line after the end of the union definition.

```
union number
{
 int n1;
 float n2;
};
union number x;
```

Accessing a union member

An individual member of a union can be accessed using the dot operator (**.**) or arrow operator(**->**). For example, the first member of the above union **x**, namely **n1** can be accessed as **x.n1**

Example program

Write a program to display size of the union elements. Use sizeof( ) operator.

```
/* size of the union elements */
#include<stdio.h>
#include<conio.h>
main()
{
 union book
 {
  int pages;
  float price;
 };
 union book bk;
 clrscr();
 printf("pages-%d\n",sizeof(bk.pages));
```

```
printf("price-%d\n",sizeof(bk.price));
printf("total-%d\n",sizeof(bk));
}
```

Output

```
pages-2
price-4
total-4
```

7. Bit Fields

Bit field provides exact amount of bits required for storage of values. If a variable value is 1 or 0 we need a single bit to store it. In the same way if the variable is expressed between 0 and 3, then the two bits are sufficient for storing these values. Similarly if a variable assumes values between 0 and 7 then three bits will be sufficient to hold the variable and so on. The number of bits required for a variable is specified by non-negative integer followed by colon.

To hold the information we use the variables. The variables occupy a minimum of one byte for char and two bytes for integer. Instead of using complete integer if bits are used space of memory can be saved. For example, to know the information about the vehicles, following information has to be stored in the memory

1. Petrol vehicle
2. Diesel vehicle
3. Two wheeler vehicle
4. Four wheeler vehicle
5. Old model
6. New model

In order to store the status of the above information we may need two bits for types of fuel as to whether the vehicle is of petrol or diesel type. Three bits for its type as to whether the vehicle is two or four wheeler. Similarly, three bits for model of the vehicle. Total bits required for storing the information would be 8 bits i.e, 1 byte.

The structure for the above problem would be as follows.

```
struct vehicle
{
 unsigned type:3;
```

```
    unsigned fuel:2;
    unsigned model:3;
   };
```

The colon(:) in the above declaration tells to the compiler that bit fields are used in the structure and the number after it indicates how many bits are required to allot to the field.

Example program

Write a program to store the information of vehicles. use bit fields to store the status of information.

```
#include<stdio.h>
#include<conio.h>
#define petrol 1
#define disel 2
#define two_wh 3
#define four_wh 4
#define old 5
#define new 6
main()
{
 struct vehicle
 {
  unsigned type:3;
  unsigned fuel:2;
  unsigned model:3;
 };
struct vehicle v;
v.type=four_wh;
v.fuel=disel;
v.model=old;
clrscr();
printf("type of vehicle:%d",v.type);
printf("\nfuel :%d",v.fuel);
printf("\nmodel:%d",v.model);
}
```

Output

```
type of vehicle:4
fuel :2
model:5
```

8. enumerated data type

The enum is a keyword. It is used for declaring enumeration types. The programmer can create his/her own data type and define what values the variables of these data types can hold. This enumeration data type helps in reading the program.

Consider the example of 12 months of a year.

enum month {jan, feb, mar, apr, may, jun, jul, aug, sep, oct, nov, dec};

This statement creates a user defined data type. The keyword enum is followed by the tag name month. The enumeration is the identifiers jan, feb, mar, apr, and so on. Their values are constant unsigned integer start from 0. The identifier jan refers 0,feb refers 1 and so on. Their values are constants unsigned integers and start from 0. The identifier jan refers to 0, feb to 1 and soon. The identifiers are not to be enclosed within quotation marks. Please also note that integer constants are also not permitted.

Example program

Write a program in C that creates an enumerated data type for 7 days of the week. Initialize the first day with 1. Get a day number from the user and display its corresponding day in words.

```
 #include<stdio.h>
#include<conio.h>
main()
{
 enum week {mon=1,tue,wed,thu,fri,sat,sun};
 clrscr();
 printf("\nmon=%d",mon);
 printf("\nwed=%d",wed);
 printf("\nfri=%d",fri);
 printf("\nsun=%d",sun);
}
```

Output

```
mon=1
```

```
wed=3
fri=5
sun=7
```

9. Typedef

The typedef is a keyword. By using **typedef** we can create new data type. The statement typedef is to be used while defining the new data type. The syntax is given below

```
typedef type dataname;
```

Here, *type* is the data type and *dataname* is the user defined name for that type.

```
typedef int hours;
```

Here, an hour is another name for int and now we can use hours instead of int in the program as follows.

```
hours hrs;
```

Example program

Write a program to create user defined data type hours on int data type and use it in the program

```
#define H 60
main()
{
 typedef int hours;
 hours hrs;
 clrscr();
 printf("Enter Hours:");
 scanf("%d",&hrs);
 printf("Minutes=%d",hrs*H);
 printf("\nSeconds=%d",hrs*H*H);
}
```

Output

```
Enter Hours:2
Minutes=120
Seconds=7200
```

Explanation

In the above example with *typedef* we have declared *hours* as an integer data type. Immediately after the *typedef* statement *hrs* is a variable of hours data type which is similar to int. further the program calculates *minutes & seconds* using hrs variable.

UNIT-V

Console I/O

1.Reading and writing characters

getchar() : it accepts one character type data from the keyboard. This function reads character type data from the standard input. It returns a single character from a standard input device (typically a keyboard). The function does not require any arguments, though a pair of empty parentheses must follow the word `getchar`.

Syntax

```
variable_name = getchar();
```

Example program

Write a program to accept characters through keyboard using getchar( ) function.

```
#include<stdio.h>
main( )
{
  char c;
  clrscr();
  printf("Enter character:");
  c=getchar( );
   printf("\nThe character is: %c",c);
}
```

Output

```
Enter character: m
The character is: m
```

putchar( ): it displays one character at a time to the Monitor. This function prints one character on the screen at a time which is read by the standard input.

Syntax

```
putchar(variable_name);
```

Example program

Write a program to print the characters using putchar( ) function.

```
#include<stdio.h>
main( )
{
  char c;
  clrscr();
```

```
  printf("Enter character:");
  c=getchar( );
  printf("The charcter is");
  putchar(c);
 }
```

Output

```
Enter character: m
The character is: m
```

2. Reading and writing strings

gets(): it accepts any line of string including spaces from the standard Input device (keyboard). gets() stops reading character from keyboard only when the enter key is pressed.

Syntax

```
gets(variable_name);
```

Example program

Write a program to accept characters through keyboard using gets( ) function.

```
#include<stdio.h>
main( )
{
  char ch[20];
  int c=0;
  clrscr();
  printf("Enter string:");
  gets(ch);
  printf("\nThe string is: %s",ch);
}
```

Output

```
Enter the string: compiler@
The string is: compiler
```

puts(): it displays a single / paragraph of text to the standard output device.

Syntax

```
puts(variable_name);
```

Example program

Write a program to print the string using puts( ) function.

```
#include<stdio.h>
main( )
{
  char ch[20];
  clrscr();
  printf("Enter string:");
  scanf("%s",ch);
  printf("The string is: ");
  puts(ch);
}
```

Output

```
Enter the string: compiler
The string is: compiler
```

The basic I/O functions

function	Operation
getchar()	Reads a character from the keyboard; usually waits for carriage return.
getche()	Read a character with echo; does not wait for carriage return; not defined by standard C, but a common extension.
getch()	Reads a character without echo; does not wait for carriage return; not defined by standard C, but a common extension.
putchar()	Writes a character to the screen.
gets()	Reads a string from the keyboard
puts()	Write a string to the screen.

3. Printf()

C provides the printf() function display the data on the standard output device (monitor). The *printf()* function is included in the header file **stdio.h**.

The general form of printf() function is

printf(“control string”, variable_list);

Where,

Control string specifies the type and format of the values to be displayed.

Variable_list is a list of variables to be displayed on the monitor.

Control string (character group)	Meaning
%c	Print a single character
%d	Print a decimal integer
%e	Print a floating point number
%f	Print a floating point number
%g	Print a floating point number
%h	Print a short int number
%i	Print a decimal or hexadecimal or octal number
%o	Print a octal number
%p	Print a pointer
%s	Print a string
%u	Print a unsigned integer
%x	Print a hexadecimal

Examples of **printf()** function are given below:

1. printf("welcome to print stamen");
2. printf("%d",number);
3. printf("%f %f",p,q);
4. printf("sum of three numbers=%d",sum);
5. printf("\n X=%d, Y=%d\n",x,y);

Integer output

The integer number can be displayed on the monitor by using %d character group. Suppose the variable var is to be displayed. And also assume that its value is 234.

Then the statement,

printf("%d",var);

Display 234 on the monitor.

The field width can also be specified for the proper format of displaying on the screen. For example,

printf("%5d",x);

if the value of x is 27631, then the output can be shown pictorially as

2	7	6	3	1

If the value of x is 678, then the output would be

		6	7	8

Here, the digits in the number are right justified (adjusted). That is , the right most digit of the number occupies the rightmost column. It can also be possible to display the number in left justified manner. That is, the left most digit of a number would occupy the leftmost column. This is carried out by placing a minus sign in the character group shown in the following printf() function

printf("%-5d",x);

if the value of **x** is **678**, then the output would pictorially be shown as

6	7	8		

If the number of digits is less than the specified field width and if you want to fill all the leading spaces by 0, then write printf() as

printf("%05d",x);

if the value of x is 678, then the output would pictorially be shown as

0	0	6	7	8

If the number of digits exceeds s the specified field width then extra memory is allocated to fit the number to be displayed. For example,

printf("%3d",number);

Suppose the content of the number is 23456. But the specified field width is only 3. Then the extra memory is allocated to fit all the 5 digits of number. Thus the output would pictorially be shown as

2	3	4	5	6

If the number to be displayed is a signed number, then a plus sign will be inserted in the character group, for example

printf("%+4d",number);

Suppose the content of the variable number is 673. Then the output will appear as,

+	6	7	3	

Floating-point Output

A floating-point number can be displayed either in a *decimal form* or a *scientific notation*. The %f and %e character groups are used respectively. For example

printf("%f",x);

if x contains the value 24, then it will appear on the standard output device as, 24.000000
we can restrict the number of digits after the decimal point by using an appropriate field width specification. For example,

printf("%6.2f",p);

This statement specifies to *print the number having a maximum of 6 characters including a decimal point and there must be two digits after the decimal point*. If the content of the variable P is 543.671, then it would be pictorially shown as

Columns → 1 2 3 4 5 6

5	4	3	.	6	7

If the **printf()** function is written as below

printf("%6.1f",p);

and for p=543.671, the output would be pictorially shown as

Columns → 1 2 3 4 5 6

	5	4	3	.	7

Again, if the above **printf()** function is written as follows ,

printf("%-6.1f",p);

Columns → 1 2 3 4 5 6

5	4	3	.	7	

Similarly, the floating point numbers in a scientific notification form can be displayed using the %e character group. For example,

printf("%e",z);

if the variable **z** contains the value 56.6732, then the output will (appear shown) as,

Columns →

1	2	3	4	5	6	7	8	9	10	11
5	.	6	6	7	3	2	e	+	0	1

If the **printf()** function is written as

printf("%10.2e",z);

This statement indicates that there are 10 columns reversed for displaying the data and there must be two characters after the decimal point. If z contains the value 56.6732, then the output will appear as

Columns →

1	2	3	4	5	6	7	8	9	10
			5	.	7	e	+	0	7

Consider another example where z takes the value 0.05 and the statement is written as

printf("%e",z);

Then, the output will be

1	2	3	4	5	6	7	8	9	10	11
5	.	0	0	0	0	0	e	-	0	2

Character Output

A single character or a string (group of characters) can be displayed onto the standard output device by using the **printf()** function. The **%c** and **%s** character groups are used for this purpose respectively. For example, consider the following **printf()** function.

printf("%c",ch);

Where, **ch** is a variable of type **char**. When this statement is executed, you will find on the screen A, if you have entered A for **ch**.

A string is displayed with the following statement.

printf("%s",book);

Here, the variable **book** is a string of characters. And before using the variable **book** it must be declared as

char book[20];

if the string to be displayed is **"programming in C"**, then it would pictorially be shown as,

p	r	o	g	r	a	m	m	i	n	g		i	n		C				

If the above printf() function is written as

printf("%20.10s",book);

If the output will be

									p	r	o	g	r	a	m	m	i	n

The field specification %20.10s is an indication to print the first 10 characters in right justified format. Similarly, a left justified string is printed by using the %-20.10s in the control string of the above printf() statement. Thus output would be

p	r	o	g	r	a	m	m	i	n									

4. scanf()

C provides a function called *scanf()* to read the values for the variable in a C program from the keyboard. The *scanf()* function is used to enter the numeric, character and string type of data. It is included in the header file **<stdio.h>**.

The general format of scanf() is as follows.

```
scanf("control string",address_list);
```

Where,

Control string is the sequence of one or more character groups. Each character group is a combination of the % symbol and one of the conversion characters. The control string specifies the type of the values which are to be supplied to the variables.

Address_list specifies the address of the memory locations where the values of input variables should be stored.

Example of scanf() statement

```
scanf("%d",&a);
scanf("%c",&ch);
scanf("%f",&f);
scanf("%s",str);
scanf("%d%d",&a,&b);
```

The below table illustrate the different character groups and the meaning associated with the scanf() statement.

Control string (character group)	Meaning
%c	Read a single character
%d	Read a decimal integer
%e	Read a floating point number
%f	Read a floating point number

%g	Read a floating point number
%h	Read a short int number
%i	Read a decimal or hexadecimal or octal number
%o	Read a octal number
%p	Read a pointer
%s	Read a string
%u	Read a unsigned integer
%x	Read a hexadecimal

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

Example 1:

```
scanf ( "%3d %3d %3d" , &a, &b, &c) ;
```

When the program is executed, three integer quantities will be entered from the standard input device (the keyboard).

Suppose the input data items are entered as

1 2 3

Then the following assignments will result:

a = 1 , b = 2 , c = 3

If the data had been entered as

123 456 789

Then the assignments would be

a = 123, b = 456, c = 789

Now suppose that the data had been entered as

123456789

Then the assignments would be

a = 123, b = 456, c = 789

as before, since the first three digits would be assigned to a, the next three digits to b, and the last three digits to c.

Finally, suppose that the data had been entered as

1234 5678 9

The resulting assignments would now be

a = 123, b = 4, c = 567

The remaining two digits (8 and 9) would be ignored, unless they were read by a subsequent scanf statement.

Example 2:

scanf("%3d %5f %c", &i,&x, &c);

If the data items are entered as

10 256.875 T

When the program is executed, then 10 will be assigned to i, 256.8 will be assigned to x and the character 7 will be assigned to c. The remaining two input characters (5 and T) will be ignored.

Example 3:

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

If the input data consisted of

a b c

(with blank spaces between the letters), then the following assignments would result:

cl = a, c2 = <blankspace>, c3 = b

If the scanf function were written as

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

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

cl = a, c2 = b, c3 = c

as intended. Note that there are some other ways around this problem. We could have written the scanf function as

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

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

File I/O

1. Definition of file

File is a set of records that can be accessed through the set of library function.

Types of File

1. Sequential file
2. Random Access file

Sequential file: in this type data's are kept sequentially. If we want to read the last record of the file we need to read all the records before that record. It takes more time. Or if we desire to access the 10^{th} record then the first 9 records should be read sequentially for reaching to the 10^{th} record.

Random Access File: in this type data can be read and modified randomly. In this type if we want to read the last records of the file, we can read it directly. It takes less time as compared to sequential file.

2. file system basics(Opening and closing a data file)

When working with a stream oriented data file, the first step is to establish a *buffer area*, where information is temporarily stored while being transferred between the computer's memory

and the data file. This buffer area allows information to be read from or written to the data file more rapidly than would otherwise be possible.

The buffer area is established by writing

FILE *fp;

Where, fp is a file pointer. Each file that we open has its own FILE structure. The information that the file contains may be its current size, its location in memory etc.

This important task is carried out by the structure FILE that is defined in **stdio.h** header file. So this file must be included. When a request is made to the operating system for opening a file, it does so by granting the request. If request is granted the operating system points to the structure FILE. In case the request is not granted it returns NULL.

fp=fopen("file name", "mode/file_type ");

example: fp=fopen("data.txt","r");

Here, **fp** is a pointer variable that contains address of the structure FILE that has been defined in the header file "stdio.h". the function fopen( ) will open a file "data.txt" in read mode. The C compiler reads the contents of the file because it finds the read mode ("r"). Here, "r" is a string and not a character.

Finally a data file must be closed at the end of the program. This can be done by the library function fclose.

fclose(ptr);

A 'C' language supports many more file handling functions that are available in standard library.

File function	Operation
fopen()	Creates a new file for read/write operations.
fclose()	Close a file associated with file pointer.
closeall()	Closes all opened files with fopen().
fgetc()	Reads a character from a file.
fputc()	Writes a character to a file.
fprintf()	Writes all types of data values to a file.
fscanf()	Reads all types of data values from a file.
putw()	Writes an integer to the file.
getw()	Reads an integer from the file
feof()	Detects the end of file.
fseek()	Sets the pointer position anywhere in the file.
ferror()	Reports error occurred while read/write operations.
Unlink()	Removes the specified file from the disk.

Modes operations (or) file_type specification

Mode	Operations
'r' (read)	Open for reading only

'w' (write)	Creates a new file for writing or open an existing file for writing
'a' (append)	Open for writing at the EOF.
r+ (read + write)	Open an existing file for reading and writing the file must exist.
w+ (write + read)	Open an empty file for reading and writing
a+ (append + read)	Open for reading and appending

Concept of Binary Files

The commonly used data files are text files. There are another kind of files also, that re known as binary files. All machine language files are actually binary files. For opening a binary file, the file mode has to be mentioned as "rb" or "wb" in fopen command.

The binary files differ from text files in two ways: The storage of newline character. The eof character. First difference is about the storage of \n, ie., newline character. In text files, \n is stored as a single newline character by user, but it takes 2 bytes of storage inside the memory.

Example

Write a program to write data to text file and read it.

```
#include<stdio.h>
#include<conio.h>
void main()
{
FILE *fp;
char c=' ';
clrscr();
fp=fopen("data.txt","w");
if(fp==NULL)
{  printf ("\n Cannot open file");
   exit(1);  }
printf ("write data & to stop press ' .':");
while (c!='.')
{  c=getche();
   fput(c,fp); }
fclose(fp);
printf("\n Contents read :");
fp=fopen("data.txt","r");
while(!feof(fp))
printf("%c",getc(fp));
}
```

Output

```
Write data & to stop press '.'
Murahari reddy.
Contents read: Murahari reddy.
```

Write a program to append data to text file and read it.

```
#include<stdio.h>
#include<conio.h>
void main()
{
FILE *fp;
char c=' ';
clrscr();
fp=fopen("data.txt","a");
if(fp==NULL)
{ printf ("\n Cannot open file");
  exit(1); }
printf ("write data & to stop press ' .':");
while (c!='.')
{ c=getche();
  fput(c,fp); }
fclose(fp);
printf("\n Contents read :");
fp=fopen("data.txt","r");
while(!feof(fp))
printf("%c",getc(fp));
}
```

Output

```
Write data & to stop press '.'
Duvvuru.
Contents read: Murahari reddy.Duvvuru.
```

3. fprintf ()

The fprintf statement has same functionality as that of printf with the only difference that fprintf stores the output in the file. This function is used for writing characters strings integers float etc. to the file. It contains one more parameter that is file pointer which points the opened file.

Example

```
# include<stdio.h>
void main()
{
file *fp;
char text(30);
clrscr();
fp=fopen ("text.txt","w");
printf("Enter text here:');
gets(text);
fprintf(fp ,"%s",text);
fclose(fp);
}
```

Output

```
Enter text here: have a nice day.
```

4. fscanf()

This function reads character, Strings integer, float etc. from the file pointed by pile pointer.

WAP to enter data into the text file and read the same use'w" file mode.

```
#include <stdio.h>
 void main()
{
 file *fp;
char text[5];
int age;
fp=fopen("Text.txt","w+");
clrscr();
printf("Name \t ge \n");
scanf("%s %d",text,&age);
fprintf(fp,"%s %d",text,age);
printf("Name \t age \n");
fscanf("fp,"%s %d",text,&age);
printf("%s \t %d\n",text,age);
fclose(fp);
}
```

Output

```
Name    Age
Rakhi   15
Name    Age
Rakhi   15
```

5. Unformatted Data Files(fread & fwrite)

Some applications involve the use of data files to store blocks of data, where each block consists of a fixed number of contiguous bytes.

For example a data file may consist of multiple structures having the same compositions or it may contain multiple arrays of the same type and size.

For such application it may be desirable to read the entire block from the data file, or write the entire block to the data file. The library functions fread and fwrite are intended to be used in situations of this type. These functions are often referred as unformatted read and write functions. Data files of this type are often referred as unformatted data files. Each of these functions requires four arguments: a pointer to the data block, the size of the data block, the number of data blocks being transferred and the stream pointer

A typical fwrite function might be written as

fwrite(&customer, sizeof (record), 1 , fpt)

A typical fread function might be written as

fread(&customer, sizeof (record), 1 , fpt)

Write a program to write and read the information about the player containing players name, age and runs

```
#include<stdio.h>
#include<conio.h>
#include<process.h>
struct record
{
char player[20];
int age;
int runs;
};
void main()
{
FILE *fp;
struct record emp;
fp=fopen("record.dat","w");
if(fp==NULL)
```

```
    {
    printf("\ncan not open the file");
    exit(1);
    }
    clrscr();
    printf("Enter player name,age &runs scored\n");
printf("=====================================\n");
    scanf("%s %d %d",emp.player,&emp.age,&emp.runs);
    fwrite(&emp,sizeof(emp),1,fp);
    fclose(fp);
    if((fp=fopen("record.dat","r"))==NULL)
    {
    printf("\n Error on openingfile");
    exit(1);
    }
    printf("\nrecord entered is\n");
    fread(&emp,sizeof(emp),1,fp);
     printf("\n %s %d %d",emp.player,emp.age,emp.runs);
    fclose(fp);
    }
```

Output:

```
Enter player name, age & runs scored
 ====================================
 Sachin 25 10000
Record entered is
Sachin 25 10000
```

6. Accessing the File Randomly (Using fseek)

Random access means that we can read the data of a particular position without reading the data before that position. This can be very useful in faster processing of the file can be updated using the random access, if we know the position of the record to be updated.

The random access is possible through use of fseek function. The fseek function is used to reposition the file pointer

Syntax:

fseek(FILE *fstream, long offset, interference_position);

We can pass three arguments through this function.

1. File pointer.
2. Negative or positive long integer number to reposition the file pointer.
3. The current position of file pointer.

Integer value	Constant	Location in the file
0	Seek_Set	Beginning of the file
1	Seek_Cur	Current position of the file point
2	Seek_End	End of the file

Example:

fseek (fp, 10, seek_set)

The file pointer is repositioned in the forward direction by 10 bytes.

Write a program to read the text file containing some sentence .use fseek() and text after skipping n characters from beginning of the file

```
#include<stdio.h>
 #include<conio.h>
#include<process.h>
void main()
{
FILE *fp;
int n,ch;
clrscr();
fp=fopen("text.txt","r");
printf("\ncontents of a file\n");
while((ch=fgetc(fp))!=EOF)
printf("%c",ch);
printf("\n How many characters including spaces would you like to skip?:");
scanf("%d",&n);
fseek(fp,n,SEEK_SET);
printf("\n information after %d bytes \n",n);
while((ch=fgetc(fp))!=EOF)
printf("%c",ch);
fclose(fp);
}
```

Output

```
Contents of a file
My name is khan
How many characters including spaces would you like to skip? :3
Name is khan
```

7. Macro

We have already seen that the #define statement can be used to define symbolic constants within a C program. The #define statement can be used for more, however, than simply defining symbolic constants. In particular, it can be used to define macros; i.e., single identifiers that are equivalent to expressions, complete statements or groups of statements. Macros resemble functions in this sense. They are defined in an altogether different manner than functions, however, and they are treated differently during the compilation process.

Consider the simple C program shown below.

```
#include <stdio.h>
#define area length * width
main( )
{
int length, width;
printf("length =") ;
scanf('%d", &length);
printf("width = " ) ;
scanf("%d" , &width);
printf( " \ n area = %d", area);
}
```

Output

```
length = 3
width = 4
area = 12
```

Macro definitions are customarily placed at the beginning of a file, ahead of the first function definition. The scope of a macro definition extends from its point of definition to the end of the file. However, a macro defined in one file is not recognized within another file.

Multiline macros can be defined by placing a backward slash (\) at the end of each line except the last. This feature permits a single macro (i.e., a single identifier) to represent a compound statement.