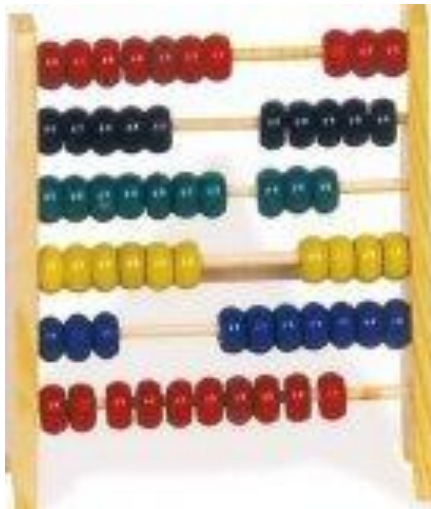


INTRODUCTION TO COMPUTER

HISTORY OF COMPUTER:

In earlier days merchants in Egypt and china used a device called as ABACUS. It was used to do all kinds of mathematical operations like addition, subtraction, multiplication, division etc., abacus was in use for around 4000 years. This was the beginning for usage of computers.



A typical structure
of abacus computer

CHARLES BABBAGE (1792 - 1871):

- He is known as FATHER OF COMPUTERS
- He is a mathematics professor who created the design for computer.
- In 1822 he created differential engine to do mathematical calculations.
- In 1833 he invented analytical engine which had 5 functional units :
 - Input unit
 - Output unit
 - Arithmetic unit
 - Control unit
 - Memory unit
- The ideas formulated by babbage in designing the analytical engine helped the scientists to lay foundation for the human's greatest achievement COMPUTER.

GENERATIONS OF COMPUTER

First generation computers : (1945 - 1956) (sample figures of first generation computers)



Machine language or binary language was the only language that was understood by computers. Binary language will be in the form of 0 and 1.

Major developments of this generation were:

- ENIAC (Electronic Numerical Integrator And Calculator)
- EDVAC (Electronic Discrete Variable And Automatic computer)
- UNIVAC (Universal Automatic Computer)
- VACCUM TUBES were used to manufacture the computers. The heat generated by vacuum tubes damaged the sensitive parts of computer. This was the main problem with vacuum tubes.
- Also machine language was very difficult to understand.

Second generation computers: (1956 - 1963)

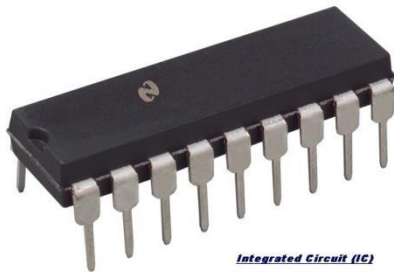


(sample figure of second generation computer)

- Second generation computer machines were based on transistor technology.
- Second generation computers were smaller as compared to the first generation computers
- The computational time of Second generation computers was reduced to microseconds from milliseconds.
- Assembly language was used to program Second generation computers. Hence, programming became more time-efficient and less cumbersome.
- Major inventions of this generation were :
 - COBOL (Common Business Oriented Language)
 - FORTRAN (FORMula TRANslator)
 - ASCII CODE (American Standard Code for Information Interchange)

Third generation computers: (1964 - 1971)

- Integrated chip (IC) was used in place of transistors for manufacturing computers.



-----> INTEGRATED CHIP (IC)

- 3rd generation computers were more smaller and cheaper than 2nd generation computers.
- PDP-8, PDP-11 IBM370 are some of the 3rd generation computers
(if possible try to search for diagrams from internet)

Fourth generation computers : (1971 - present)

Major developments of this generation are :

- Personal computer (PC)
- Graphical User Interface (GUI)
- Computers were designed with MICRO PROCESSOR technology.

Fifth generation computers : (future)

- Aim of computer experts is to develop ARTIFICIAL INTELLIGENCE (AI) have all qualities of humans.

BASIC TERMS

Data: it is a collection of raw facts.

Information: it is a processed data(meaningful data). It is the result of processing the data.

Input: the data and information entering (given by user) into a computer.

Output: the resultant information obtained by a computer.

Instruction: is to process the given data.

Program: is a sequence of instructions that can be executed by the computer to solve the given problem.

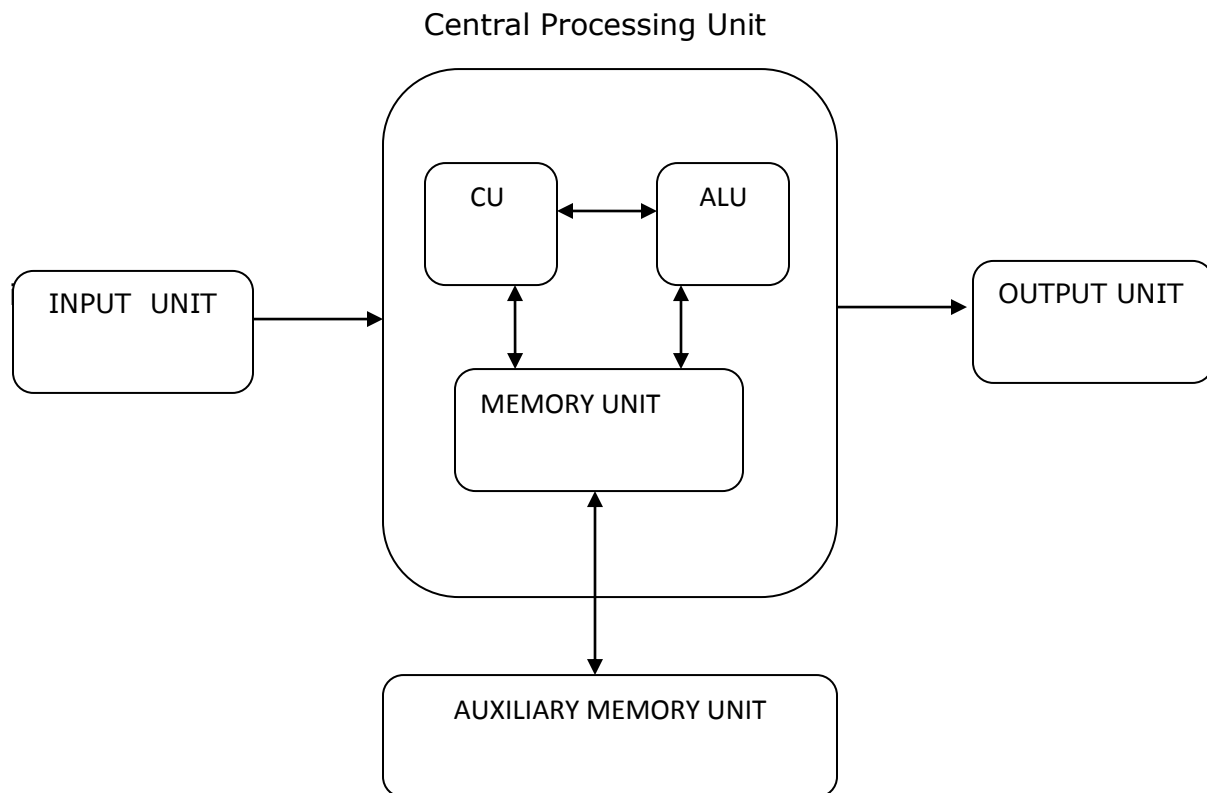
COMPUTER

A computer is a electronic device which accepts data as input, process it according to instructions given, and displays the output.

Characteristics of a computer :

- Huge data storage (large amount of data can be stored)
- High- speed calculations (calculation are done quickly)
- Accuracy (correct result is produced)
- Reliability (gives consistent results even though it is based on electronic circuits)
- Diligence (won't get tired even it works continuously for many hours)

BLOCK DIAGRAM OF COMPUTER



1. INPUT UNIT:

It obtains data from various input devices and place this information at the disposal of the other units so that information may be processed. Most information is entered into computer today through keyboard and mouse devices. Information can also enter by speaking and by scanning images. Some of other input devices are touch screen (used in ATM's), joystick, electronic pen etc. unit

2. CENTRAL PROCESSING UNIT:

The CPU performs functions similar to the Brain of a human body. It consists of three components :

- Control unit (CU)
- Arithmetic and Logical unit (ALU)
- Memory unit

Control Unit :

The control unit directs all operations inside the computer. It is known as nerve centre of the computer because it controls and coordinates all hardware operations i.e. those of the CPU and input-output devices. Its actions are

- It fetches the required instructions from the main storage.
- It also transfers the results from ALU to the memory and onto the output device for printing.
- It gives command to transfer data from the input device to the memory to ALU.

Arithmetic and Logic Unit (A.L.U):

It operates on the data available in the main memory and sends them back after processing, once again to the main memory A.L.U performs two functions

- 1) It carries out arithmetical operations like addition, subtraction, multiplication and division .
- 2) It performs certain logical actions based on AND and OR functions.

3. Memory Unit :

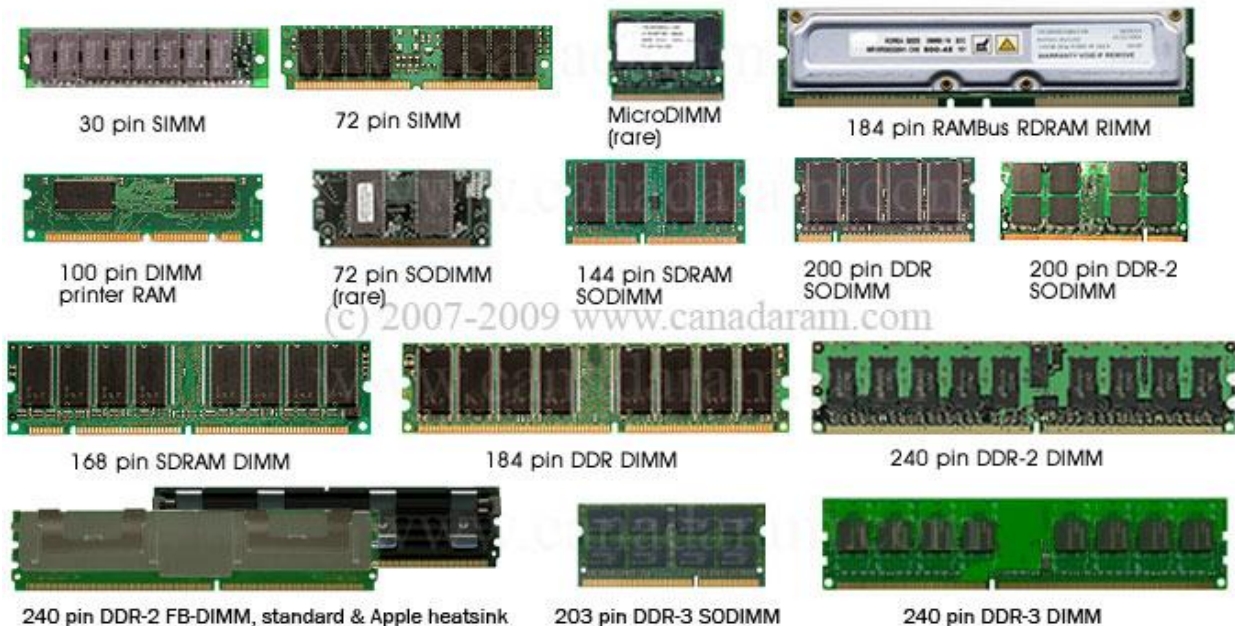
It is also called as main memory / primary memory because, like the human memory it is able to store information, which can be recalled or accessed when required. The program of instructions has to be stored in the main memory in order to make it work automatically.

Types of main memory are :

- ➔ Random Access Memory (RAM)
- ➔ Read Only Memory (ROM)

Random Access Memory (RAM) : It is a volatile memory i.e., data cannot be stored permanently. All the data entered into system are directly stored into RAM. Users can READ from or WRITE into this memory. Once the system is turned off, these contents will be erased.

Note, as well as the different number of pins, the different spacing of the slots in the connector-edge



Read Only Memory (ROM) : It is non-volatile memory i.e., data can be stored permanently. The contents of ROM can only be read but cannot be changed. Some variations of ROM are :-

- **PROM (Programmable read only memory)**
 - Contents of this memory are decided by user. Once the chip is programmed the recorded instructions are permanently loaded during manufacturing of computer.
- **EPROM (Erasable programmable read only memory)**
 - Similar to PROM but contents of this memory can be changed by exposing his memory to ultra-violet rays.
- **EEPROM (Electrically Erasable programmable read only memory)**
 - Contents of this memory can be erased electrically and later new information can be stored.

4. Output Unit :

The result of any computer processing has to be communicated to the user. Output devices translate the computers output into a form understandable to human beings. Some of the output devices are display screen, printer ,speakers etc.,,,

5. Auxiliary Memory Unit :

It is also called as SECONDARY MEMORY / EXTERNAL STORAGE. These are the devices that hold the mass of information, which may be transferred during processing. These devices are used for permanent storage of data. As compared to the primary memory, it has a much larger capacity, but is not as fast. The computer thus takes slightly more time to retrieve from secondary storage.

Example: Hard disk, Compact disks (CDs)

COMPONENTS OF COMPUTER

The major components of computer are **HARDWARE** & **SOFTWARE**.

HARDWARE:

It is a general term used to represent the physical and tangible components of the computer itself i.e. those components which can be touched and seen. It includes

- (1) Input devices
- (2) Output devices
- (3) Central processing Unit
- (4) Storage devices

SOFTWARE:

It is a general term to describe all the forms of programs associated with a computer. Without software, a computer is like a human body with out a soul.

It is a general term to describe all the forms of programs associated with a computer. Without software, a computer is like a human body without a soul.

The classifications of software are :-

- System software
- Application software

System software includes :-

- i) Operating system
- ii) Language Translators

OPERATING SYSTEM is the interface between the user and the computer hardware.

Ex: MS-DOS, Solaris ,Unix , Linux.

LANGUAGE TRANSLATOR is a software that accepts input in one language and convert them into another language.

The types of translators are :

- Compiler
- Assembler
- Interpreter

Application software is a set of programs that allows the computer to help users in solving problems.

Ex:- MS-OFFICE , web browsers etc.,,

OPERATING SYSTEM (OS)

An ***operating system*** is a set of programs which acts as an interface between the user of computer and the computer hardware. User or application programs uses the hardware indirectly through operating system

The OS controls and co- ordinates the use of hardware among the application programs

The primary aim of OS is to provide convenience to the user in using the system.

The secondary aim is to use the system hardware in an efficient way.

The OS is used to manage the various resources and operations of the computer system. Depending on the manufacture, the OS is called by the different names such as *monitor, supervisor, executive, kernel*.

The major functions of OS are:

- Scheduling and loading of programs in order to provide a continuous job processing sequence.
- Controlling the hardware resource such as I/O devices, secondary storage device etc.
- Protecting hardware ,software, and data from the improper use.
- Memory and CPU time management.

- File and software management.
- Facilitates easy communication between the computer and the user.

A typical OS will have the following modules

a)**CPU Schedules**: it allocates CPU to the various resources such as I/O devices, memory devices etc.,

b)**Device Manger** : It makes devices functional .it manages various input and output devices that are interfaced with the system at later date.

c)**File Manager** : it provides facilities to the users for using the system efficiently.It also makes the usage of system easy.

d)**Other Utilities**: For information maintenance such as text editor, copy file from one media to another ,make the various modules usable etc.

Evolution of OS :

1.Batch Processing:several jobs are stacked together and processed them in groups(batches) for efficient operation. several programs run one after another without need of human operator to run each program individually

2.Multi Programming or Multitasking:Multiprogramming refers to the interleaved execution of two or more different and independent programs by the same computer.in this technique ,more than one program are kept in the memory. The OS starts executing a job .when the job needs to perform an I/O operation OS switches the CPU to next job.when the job waits,CPU is switched to execute another job, and so on.Finally,the first job will have finished waiting and will get the CPU back ,As long as there is some job to complete, the CPU will never be idle.

3.Time sharing:A time shared OS allow several users to share the computer on time basis. Many programs reside in main memory. The CPU allots a fixed time slice or quantum to each job. when the time slice of a job expires or the job need to perform an I/O operation ,the OS switches the CPU to next job in the queue .As the CPU switches rapidly from one job to the next job ,users of the computer feel that they are working independently while one computer is shared among many user actually.

4.Multi Processing: A multiprocessing system is one in which more than one processor are linked together in a co coordinated way.They share main memory and I/O devices. They can also execute portions of the same program

1. Compiler

- A compiler is a program which translates a high level language program into machine language of the host computer.
- It translates all the instructions of HLL program at a time. A compiler also produces the error messages that occurs during translation.
- Once a program is error free , then the compiled software is not needed in the memory because the machine code itself can be used again and again.

Ex: Pascal complier , C Complier, COBOL Complier.



2. Interpreter

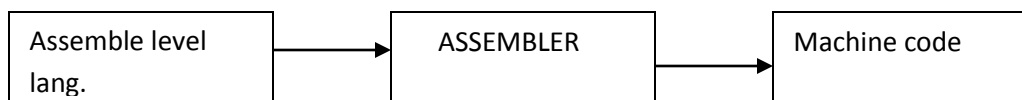
- A interpreter is a program which translate s a high level language program into machine language of the host computer.
- It translates each instruction and executes it immediately. If there is any error in any line ,it shows the error message at the same time and program execution cannot continue until the error is rectified.
- Once a program is error free ,the memory is unnecessarily occupied by interpreter as it has to reside for every run.



3. Assembler

An assembler is program which translates an assembly language program into the machine language. It gives one machine language instruction for each assembly language instruction. It also produces error messages .

Ex: 8085 assembler,8086 assembler



Differences between Compiler and Interpreter

Compiler	Interpreter
Popular for developed programs.	Popular in program development environment.
Converts entire code at a time.	Converts the code line by line.
Creates a permanent object file.	No object file is created.
Reports all the errors at a time.	It stops at the first error in program.

TYPES OF PROGRAMMING LANGUAGES

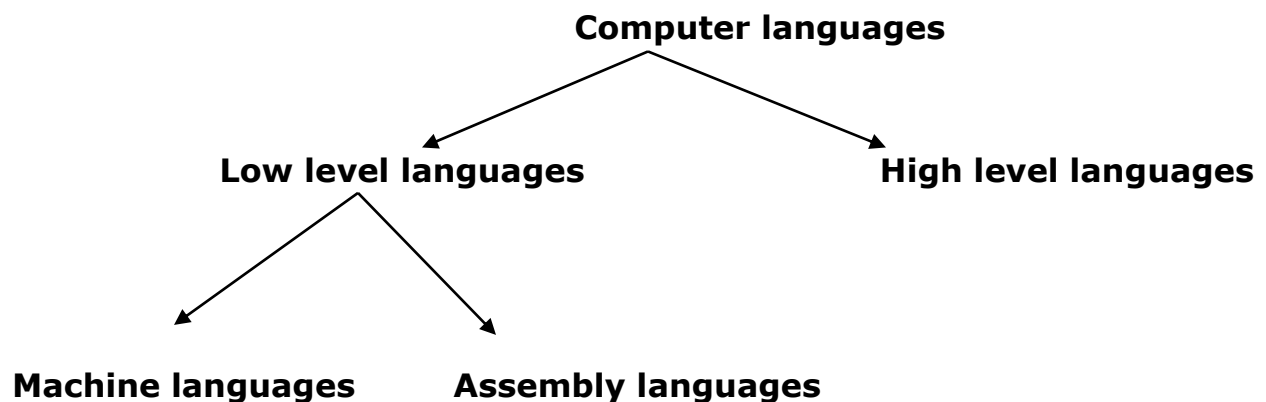
A language is meant for communication purpose

Natural language like English, Hindi, Telugu etc., are generally used by human beings to communicate with each other.

With these languages we can express or exchange our ideas.

Similarly a computer language is used for communication between human being and the computer.

A program language can be defined as **a language used for expressing a set of computer instructions called program**



MACHINE LANGUAGE

MLL can be defined as **a set of instructions coded so that the computer can use it directly without further translation**

Advantages of machine Language:

- 1.Computer understands only the machine language
- 2.Programs run very fast

Limitations of Machine Language:

- 1.Programmer has to remember the following
 - a)Binary equivalent to the OP code
 - b)Binary address of the memory location being used
- 2.Programmer written in machine language are very lengthy
- 3.Difficult to correct /modify programs
- 4.Machine language is machine dependent. That is program written for a particular computer has to be changed for using on a different computer.

ASSEMBLY LEVEL LANGUAGE:

Each instruction of assembly level language contains

- a) symbolic operation code and
- b) symbolic address

Ex:

ADD is mnemonic op code for addition

SUB is mnemonic op code for subtraction

LOAD is mnemonic op code for loading

STORE is mnemonic op code for storing

Suppose to add two number stored in locations B and C and store the result in storage location D

assemble language program as follows:

LOAD B load the contents of B into accumulator

ADD C add the contents of C to accumulator

STORE D store the result (contents of accumulator)into storage location D

Advantages of assembly Language:

1. Program correction /modification is relatively easier
2. Less time is required in writing a program than machine language
3. Language is simple when compared with machine language

Limitations of assembly Language:

- 1.Execution of program takes more time as it has to be converted into machine language

HIGH LEVEL LANGUAGES(HLL)

These are English like languages that are close to our native language.

Ex: BASIC, COBOL, FORTRAN, PASCAL, C etc.,

Suppose to add two number stored in locations B and C and store the result in storage location D

Pascal : $D = B + C;$

Cobol : ADD B to C GIVING D

C : $D = B + C$

Advantages of High level Language:

- 1.The High level Language are easier to use and understand than the machine languages and the assembly language
2. The programs written in HLL are much more compact and self explanatory than their low level counterparts
3. High level Language are provide a better documentation than low level languages.
4. High level Language are machine independent
5. 2.Program correction /modification is much easier than low level languages.

Limitations of High level Language:

- 1.Less efficient as they take more execution time when compared to low level languages
- 2.The programmer cannot completely connect the total power available at hardware level .

ALGORITHM

A step-by-step problem-solving procedure for solving a problem in a finite number of steps is known as algorithm.

Properties of the algorithm are :-

1) Finiteness: - an algorithm terminates after a finite number of steps.

2) Definiteness: - each step in algorithm is unambiguous. This means that the action specified by the step cannot be interpreted (explain the meaning of) in multiple ways & can be performed without any confusion.

3) Input:- an algorithm accepts zero or more inputs

4) Output:- it produces at least one output.

5) Effectiveness:- it consists of basic instructions that are realizable. This means that the instructions can be performed by using the given inputs in a finite amount of time.

Example : algorithm to find the sum of 'n' integers

Step-1 : begin

Step-2 : initialize variables $s=0$ and n

Step-3: enter the number integers

Step-4: enter the elements

Step-5: calculate sum using $s = n*(n+1)/2$

Step-6: print n value

Step-7: end

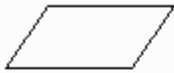
FLOWCHART

Pictorial representation of an algorithm is known as flowchart.

Different symbols used in flow chart are :



An oval is used to indicate the beginning or end of an algorithm.



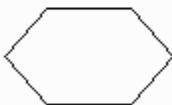
A parallelogram indicates the input or output of information.



A rectangle indicates a computation, with the result of the computation assigned to a variable.



A diamond indicates a point where a decision is made.



A hexagon indicates the beginning of the repetition structure.



A double lined rectangle is used at a point where a subprogram is used.



An arrow indicates the direction of flow of the algorithm. Circles with arrows connect the flowchart between pages.

ADVANTAGES OF USING FLOWCHARTS

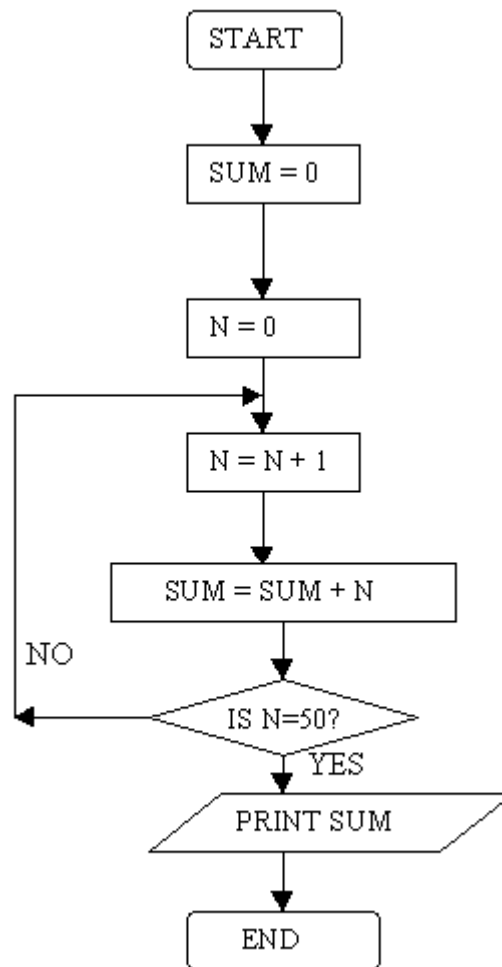
The benefits of flowcharts are as follows:

1. **Communication:** Flowcharts are better way of communicating the logic of a system to all concerned.
2. **Effective analysis:** With the help of flowchart, problem can be analysed in more effective way.
3. **Proper documentation:** Program flowcharts serve as a good program documentation, which is needed for various purposes.
4. **Efficient Coding:** The flowcharts act as a guide or blueprint during the systems analysis and program development phase.
5. **Proper Debugging:** The flowchart helps in debugging process.
6. **Efficient Program Maintenance:** The maintenance of operating program becomes easy with the help of flowchart. It helps the programmer to put efforts more efficiently on that part

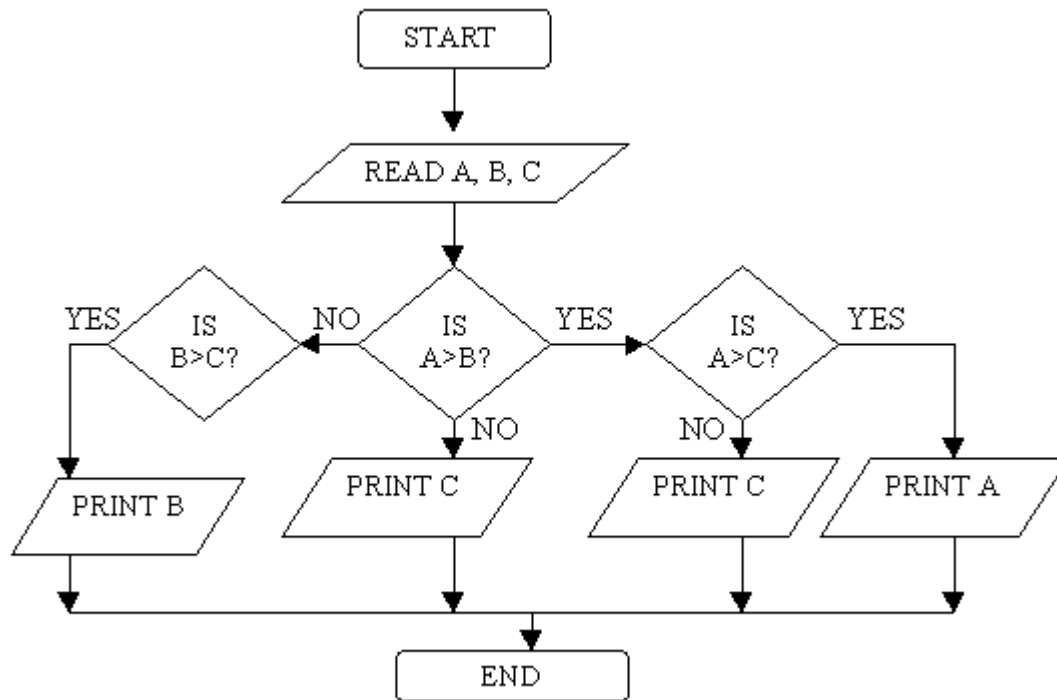
LIMITATIONS OF USING FLOWCHARTS

1. **Complex logic:** Sometimes, the program logic is quite complicated. In that case, flowchart becomes complex and clumsy.
2. **Alterations and Modifications:** If alterations are required the flowchart may require re-drawing completely.
3. **Reproduction:** As the flowchart symbols cannot be typed, reproduction of flowchart becomes a problem.
4. **The essentials of what is done can easily be lost in the technical details of how it is done.**

Flow chart for finding the sum of first 50 numbers



Flowchart for finding maximum of 3 numbers



NOTE :

Practice writing algorithms for as many programs as possible.

NUMBER SYSTEM

A number system of base (or radix), r is a system that uses distinct symbols for r digits. To determine the quantity that the number represents, it is necessary to multiply each digit by an integer power of r and then form the sum of all weighted digits.

They are of four types:

1. Decimal number system.
2. Binary number system.
3. Octal number system.
4. Hexa decimal number system.

Decimal number system:

The decimal number employs the radix – 10.

Ex:- $724.5_{(10)}$ is represented as

$$\begin{array}{ccccccc} 7 & 2 & 4 & . & 5 & & \\ & 10^2 & 10^1 & 10^0 & 10^{-1} & & \\ = 7*10^2 + 2*10^1 + 4*10^0 + 5*10^{-1} & = & 700+20+4+0.5 \end{array}$$

Binary number system:

The binary number system uses the radix 2. The two digit symbols used are 0 and 1.

Ex:- $1101_{(2)}$ is represented as

$$1 \quad 1 \quad 0 \quad 1 \quad = 1*8 + 1*4 + 0*2 + 1*1 = 13_{(10)}$$

$$2^3 \quad 2^2 \quad 2^1 \quad 2^0$$

Octal number system:

The octal number system uses the radix 8. The eight numbers used in octal system are 0-7.(i.e, 0,1,2,3,4,5,6,7).

Ex:- $34.6_{(8)}$

$$3 \quad 4 \quad . \quad 6$$

$$8^1 \quad 8^0 \quad 8^{-1}$$

$$= 8 * 3 + 1 * 4 + 6 * 0.125 = 24 + 4 + 0.75 = 28.75_{(10)}.$$

Hexa decimal number system:

The hexa decimal number system uses the radix 16. The sixteen numbers used in hexa decimal system are 0-F.(i.e., 0,1,2,3,4,5,6,7,8,9,A-10,B-11,C-12,D-13,E-14,F-15).

Ex:- $F3_{(16)}$

$$F \quad 3 \quad = \quad 15 * 16 + 3 * 1 \quad = \quad 240 + 3 \quad = \quad 243_{(10)}.$$

$$16^1 \quad 16^0$$

CONVERSIONS

Decimal to Binary:

$$\begin{array}{lcl}
 1) & 36 & \\
 & 2 \mid 36 & \\
 & 2 \mid 18 - 0 & \\
 & & \\
 & 2 \mid 9 - 0 & \\
 & & \\
 & 2 \mid 4 - 1 & \\
 & & \\
 & 2 \mid 2 - 0 & \\
 & & \\
 & \mid 1 - 0 &
 \end{array}$$

So binary equivalent of $36_{(10)}$ is $(100100)_{(2)}$

$$\begin{array}{lcl}
 2) & 41 & \\
 & 2 \mid 41 & \\
 & 2 \mid 20 - 1 & \\
 & & \\
 & 2 \mid 10 - 0 & \\
 & & \\
 & 2 \mid 5 - 1 & \\
 & & \\
 & 2 \mid 2 - 1 & \\
 & & \\
 & \mid 1 - 0 &
 \end{array}$$

So binary equivalent of $41_{(10)}$ is $(101001)_{(2)}$

Floating point to Binary:

$$\begin{array}{rcl} \mathbf{1)} \quad 36.12_{(10)} & & \\ & 2 \mid 36 & \\ & 2 \mid 18 - 0 & \\ & & \\ & 2 \mid 9 - 0 & \\ & & \\ & 2 \mid 4 - 1 & \\ & & \\ & 2 \mid 2 - 0 & \\ & & \\ & \mid 1 - 0 & \end{array}$$

For decimal value 0.12

$$0.12 * 2 = 0.24$$

$$0.24 * 2 = 0.48$$

$$0.48 * 2 = 0.96 \quad \text{so binary value for decimal place is taken till}$$

$$0.96 * 2 = 1.92 \quad 1.92 - \text{i.e., } (0001)$$

$$0.92 * 2 = 1.84$$

$$0.84 * 2 = 1.68$$

So binary equivalent of $36.12_{(10)}$ is $(100100.0001)_{(2)}$

$$\mathbf{2)} \quad 14.625_{(10)} \quad \quad 2 \mid 14$$

$$2 \mid 7 - 0$$

$$2 \mid 3 - 0$$

$$\mid 1 - 1$$

$$0.625 * 2 = 1.25$$

$$0.25 * 2 = 0.50$$

$$0.5 * 2 = 1.0 \quad \text{so binary value for decimal place is taken till}$$

$$\text{Last - i.e., } (101)$$

So binary equivalent of $14.625_{(10)}$ is $(1100.101)_{(2)}$

Binary to Decimal:

$$1) (100100)_{(2)}$$

$$1 \quad 0 \quad 0 \quad 1 \quad 0 \quad 0$$

$$2^5 \quad 2^4 \quad 2^3 \quad 2^2 \quad 2^1 \quad 2^0$$

$$1 * 32 + 0 + 0 + 1 * 4 + 0 + 0 = 32 + 4 = 36_{(10)}.$$

$$2) (100001)_{(2)}$$

$$1 \quad 0 \quad 0 \quad 0 \quad 0 \quad 1$$

$$2^5 \quad 2^4 \quad 2^3 \quad 2^2 \quad 2^1 \quad 2^0$$

$$1 * 32 + 0 + 0 + 0 + 0 + 1 * 1 = 32 + 1 = 33_{(10)} .$$

Octal number to Binary number:

Convert each number to Binary format using 3 Binary digits.

$$1) 72_{(8)} - (111010)_{(2)}$$

$$2) 64_{(8)} - (110100)_{(2)}$$

Hexa Decimal number to Binary number:

Convert each number to Binary format using 4 Binary digits.

$$1) AF_{(16)} - (1015) - (10101111)_{(2)}$$

$$2) A3_{(16)} - (103) - (10100011)_{(2)}$$

Hexa Decimal number to Decimal number:

$$1) 30_{(16)} - 3 * 16^1 + 0 = 48 + 0 = 48_{(10)}$$

$$2) B65F_{(16)} - 11 * 16^3 + 6 * 16^2 + 5 * 16^1 + 15 * 16^0 = 46687_{(10)}$$

Octal number to Decimal number:

$$1) 630_{(8)} - 6 * 8^2 + 3 * 8^1 + 0 + 4 * 8^{-1} = 384 + 24 + 0 + 0.5 = 408.5_{(10)}$$

Binary number to Octal number:

$$1) (101\ 100\ 011\ 010\ 111 . 111\ 100\ 000\ 110)_{(2)} = 54327.7406_{(8)}$$

5 4 3 2 7 7 4 0 6

$$2) (11\ 100 . 1\ 001)_{(2)} = 34.11_{(8)}$$

3 4 1 1

Binary number to Hexa Decimal number:

$$1) (0101\ 1111\ 0001\ 1010 . 1110\ 0011)_{(2)} = 5F1A . E3_{(16)}$$

5 F 1 A E 3

$$2) (101\ 1010\ 1110 . 1100\ 0010)_{(2)} = 5AE . C2_{(16)}$$

5 A E C 2

HISTORY OF C

- 'C' language was evolved from two previous languages. BCPL and B.
- 'C' is developed by Dennis Ritchie at bell laboratories and was originally implemented on a PDP-11 computer in 1972.
- C uses many of the important concepts of BCPL and B while adding data typing and other powerful features.
- C initially became widely known as the development language of the UNIX operating system.
- Today all new major operating systems are written in C and C++.
- 'C' is a hardware independent. With careful design it is possible to write C programs that are portable to most computers.

CHARACTERISTICS OF C-LANGUAGE

1. **MODULARITY**: means ability to break down large modules into sub-modules. The programming language which supports modularity is called as modular programming language.
2. **EXTENDIBILITY**: ability to extend the already existing software by adding new features to it. C software can be extended.
3. **PORTABILITY**: it is the ability to install (port) the existing software in different platforms. C software can be installed in any platform.
Ex: - Windows platform, Linux platform.
4. **SPEED**: programs written in C are very efficient and fast. This is due to variety of data types and operators available in C. There are only 32 keywords in C and they form the basic building of C-language.

WHY C LANGUAGE CALLED MIDDLE LEVEL LANGUAGE ?

- Though 'C' is a high level language, it is often referred to as **middle level language** because programs written in 'C' language run at the speeds matching to that of programs written in assembly language.
- The speed of the 'C' language programs is very fast. That's the reason 'C' language is used in system software's (programs interact with the hardware e.g.. device drivers and operating systems).

GENERAL FORMAT OF A 'C' PROGRAM

Documentation section

Link section

Definition section

Global declaration section

main() function section

{

Declaration part

Executable part

}

Sub-program section

Function1();

Function2();

.

.

Function n();

Documentation Section : To increase the readability of the program the programmer should write the appropriate explanation regarding the statements in the program. These explanatory lines are known as COMMENTS and are not executed by the compiler. Comments are written as given below:-

/*comments.....*/

Comments can be written anywhere in the program.

Link Section : This section contains the header files which contains library functions.They can be declared as:-

```
#include< header-file>
```

By writing a statement in the format as shown above, we are calling the compiler to include all the respective functions present in a header file into our program.

```
Ex: #include<stdio.h> /* this header file includes the printf() and scanf()*/  
    #include<conio.h> /* this header file includes clrscr() and getch() */
```

Definition section : This section is to define constants in the program and their value cannot be changed throughout the execution of the program.

Syntax: `#define constant_name constant_value;`

```
Ex: #define PI 3.14;
```

`/* In the program wherever we are using PI the constant value 3.14 will be replaced and cannot be changed. */`

Global declaration section : The variables declared above main() are known as global variables and these variables can be accessed anywhere in the program.

main() :

- Execution of every c-program starts with main() function.
- It must be written in lower-case letters.
- There must be one and only one main() in a c-program
- Any program in c cannot be executed without main() function.

Curly braces:

- { indicates the beginning of the program or a function.
- } indicates end of the program or function.

Declaration part: In this section, variables of different data types are declared and may be initialized.

Syntax: `datatype variable name;`

```
Ex:- int a; /* variable 'a' of integer datatype is declared */
```

Executable part: The sequence of steps to be executed (logic), in order to solve a particular program will be written here and the compiler will process the input data according to the steps written by the programmer and displays the output .

Sub-program section : The series of tasks that are to be performed by the main() are divided into individual modules / sub-programs which are known as FUNCTIONS. Each function will perform a particular task and returns a value to main() if specified by the programmer.

BASICS OF TYPICAL 'C' PROGRAM DEVELOPMENT ENVIRONMENT

C programs typically go through six phases to be executed .These are

- **Edit**
- **preprocess**
- **compile**
- **link**
- **load** and
- **execution**

All these phases should be written in the same sequence

Edit: The first phase consists of editing a file. This is accomplished with an editor program. The programmer types a C program with the editor and makes corrections if necessary, then storing the program on a secondary storage device such as a disk. C program file names should end with the **.C** extension.

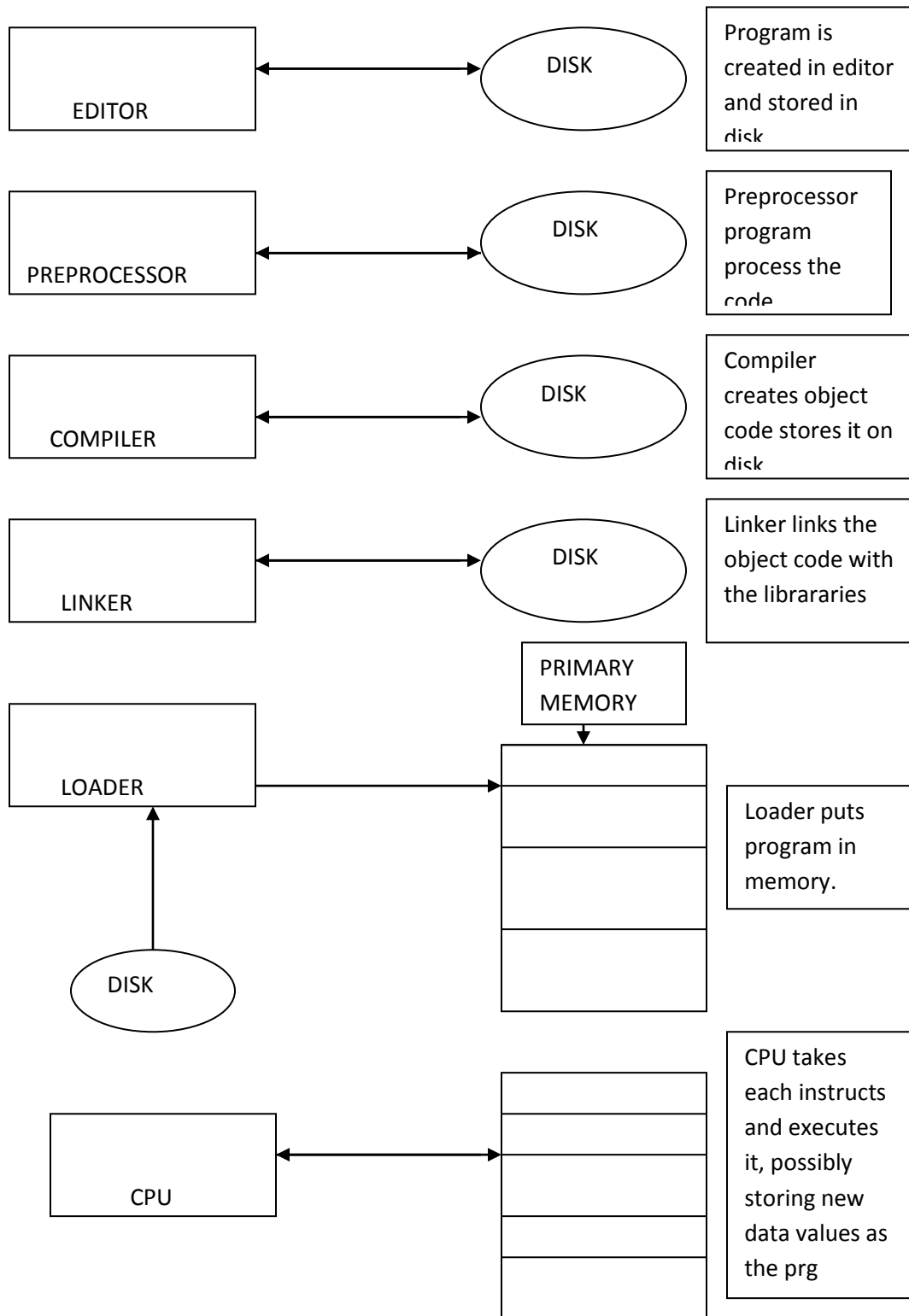
Preprocess: The preprocessor is automatically invoked by the compiler before the program is converted to machine language. A preprocessor program executes automatically before the compiler's translation phase begins.

Compile: Next, the programmer gives the command to compile the program. The compiler translates the C program into machine language code (also referred to as object code).

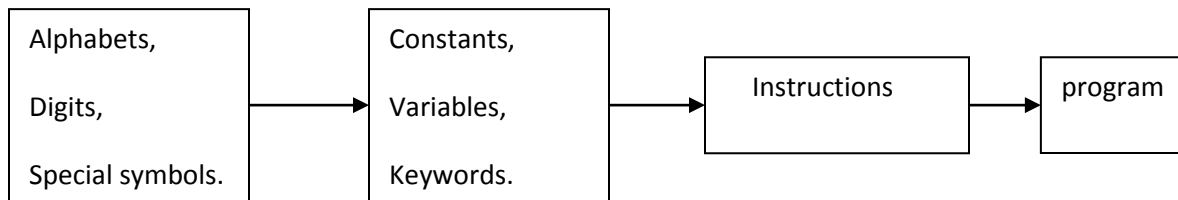
Link: Linker links the object code with the required functions to produce an executable code.

Load: The next phase is loading. Before a program can be executed, the program must first be placed in memory. This is done by the loader, which takes the executable code from disk and loads it into the memory. Additional components from shared libraries that support the program are also loaded by the loader.

Execution: Finally, the computer under the control of the CPU executes the program one instruction at a time. After executing the program we get the required output.



STEPS IN LEARNING C-LANGUAGE :



CHARACTER SET IN C-LANGUAGE :

A Character denotes any alphabet, digit or special symbol used to represent information. There are 255 characters defined in c. Every character is represented with an **ASCII** value. ASCII stands for AMERICAN STANDARD CODE FOR INFORMATION INERCHANGE.

CHARACTERS	SYMBOLS	ASCII VALUES
ALPHABETS	A,B,C.....Z	65-90
	a,b,c,.....z	97-122
DIGITS	0,1,2,3,.....9	48-57
SPECIAL SYMBOLS	~,!,@,#,\$,%^^,&*,(,),_+, ,`,>,<=,~,?/,\\,}{,[,] :,,, etc	0-47,58-64,91-96 123-127

C – TOKENS

In a passage of text, individual words and punctuation marks are known as tokens. Similarly, in C , the smallest individual units are called as **C-TOKENS**.

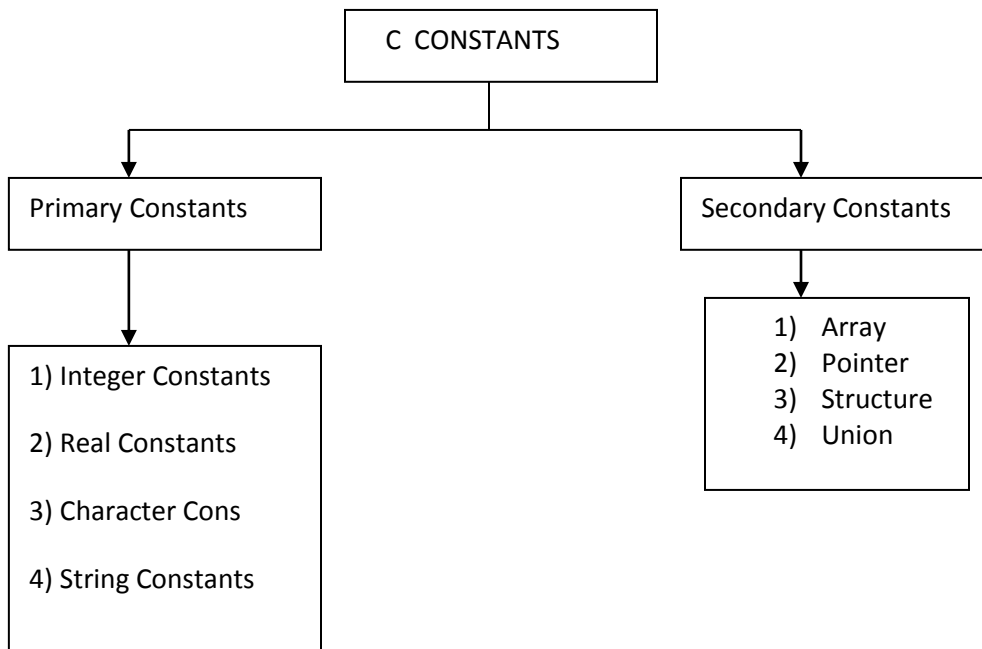
There are six types of c-tokens:-

- Keywords
- Identifiers (variables)
- Constants
- Special symbols
- Strings
- Operators

CONSTANTS: are referred to fixed values whose value do not change during the execution of c –program

C constants are divided into two categories:

- (1) Primary constants
- (2) Secondary constants



INTEGER CONSTATNTS

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

Valid-Ex: 426

+789

-3456

Invalid Ex:- 546.89 (decimal point)

10,000 (comma is present)

20 000(blank is present)

REAL CONSTANTS

- a) A Real constants must have at least one digit.
- b) It may or may not have a decimal point.
- c) It can be either positive or negative.
- d) If no signs precede it considers as positive no.
- e) No commas or blanks are allowed within a real constant.

EX: 325.345

-345.322

CHARACTER CONSTANTS

A character constant is a single alphabet, a single digit or a single special symbol enclosed with in ` `(single inverted commas).

- a) The maximum length of a character constant can be 1 character. If more than 1 characters it is not a character constant.

Valid Examples: `a` Invalid : `ab`

`1` `34`

`%` `2a`

`&`

Note: Character constant stored in 1 byte

STRING CONSTANTS:

A single character or group of characters enclosed in "" (double quotes) is called as string constant.

Examples: "a" "ab" "Hello" "2345"

NOTE: String constants stored in 2 bytes.

REPRESENTATION OF NUMBERS

INTRODUCTION:

Language is used for communication.

Language → HLL → english-like language/human understanding language.

→ LLL → machine language. (binary format ie, 0 and 1)

1. The english like language which we use may consists of:-
 - a) Alphabets,
 - b) Numbers,
 - c) Special symbols.
 2. Each of the above is assigned with a specific number called as ASCII code.
Ex: - a → 97, A → 65, etc.
 3. Whatever the code, we write on system is converted to binary format(0,1).
 4. All these binary format are stored in registers.
 5. Register is a collection of flip-flops.
Flip-flop is the smallest cell, which can hold a value.(0→ off state, 1→ on state).
Ex: - 4-bit register is a combination of 4 flip-flops.
- Number representation in binary format, can be expressed in 2ways:
- a) FIXED-POINT REPRESENTATION,
 - b) FLOATING-POINT REPRESENTATION.

FIXED-POINT REPRESENTATION: - means the position of decimal point is fixed.

Ex: - 0.5, 0.1, etc.

FLOATING-POINT REPRESENTATION: - means the position of decimal point is changing as per requirement.
Ex: - $1/3 = 0.3333 = 3.333 \times 10^{-1} = 33.33 \times 10^{-2}$ etc.

FIXED-POINT REPRESENTATION

- If in a register (r1), if the decimal position is fixed at right most position → value is an integer.
 - If in a register (r1), if the decimal position is fixed at left most position → value is an fraction.
- Ex: - 10.25
- | | |
|---------|----------|
| R1 | R2 |
| 10. | .25 |
| Integer | Fraction |
- The representation / manipulation in arithmetics or any processing of data in a register(computer) is done in the following manner.
 - a) Whenever we give a number without representing its sign, by default it is positive.
 - b) As, in computer, it performs only addition.(-, *, /, % operations done only in form of addition).

Ex: - $A + B = A + B.$

$A - B = A + (-B).$

So, we need to have separate representation for positive and negative numbers.

REPRESENTATION OF THE POSITIVE NUMBERS

$$+2 = 0010 \text{ (4-bit repn).}$$

REPRESENTATION OF THE NEGATIVE NUMBERS

1. Representing a negative number or a number in negative form is of 3 types: -

- SIGNED-MAGNITUDE REPN:

The magnitude is represented in the same form and sign bit is '0' for positive and '1' for negative.

$$\begin{array}{rcll} \text{Ex: -} & +2 & = & 0 \quad 010 \\ & -2 & = & 1 \quad 010 \end{array}$$

When we go for signed repn, left most bit represents sign and remaining bits give magnitude.

Ex: - 4-bit repn

$$\begin{array}{lcllcl} \text{Unsigned} & : & \text{highest value} & = & 15 & = & 1111 \\ \text{Signed} & : & \text{highest value} & = & 7 & = & 0/1 \quad 111 \end{array}$$

- SIGNED-1'S COMPLEMENT REPN:

As, we need to have separate repn for negative and positive numbers, we take 1's complement form for negative numbers.

$$\begin{array}{rcll} +2 & = & 0 & 010 \\ -2 & = & 1 & 101 \quad (\text{1's complement form - changing 1 to 0 and vice-versa}). \end{array}$$

- SIGNED-2'S COMPLEMENT REPN:

When we have arithmetic calculation for zero, we need to have the same repn for +0 and -0. So, to have same repn for both, we take 2's complement form for negative numbers.

$$\begin{array}{rcll} \text{Ex: -} & +2 & = & 0 \quad 010 \\ & -2 & = & 1 \quad 101 \quad (\text{1's complement repn}) \\ & & & \quad \quad \quad \underline{+1} \\ & -2 & = & 1 \quad 110 \quad (\text{2's complement repn}) \end{array}$$

ARITHMETIC ADDITION: -

$$3 + 2 = 5;$$

		Sign	magnitude	
3	=	0	011	
2	=	0	010	
5	=	0	101	(0 - positive, 101 - 5)

ARITHMETIC SUBTRACTION: -

$$+3 - 2 = +1;$$

		Sign	magnitude	
+3	=	0	0 1 1	
-2	=	1	1 1 0	(take -2 in 2's complement repn)
+1	=	0	0 0 1	(0 – positive, 0 0 1 - 1)

Carry is discarded, as we are taking 4-bit register.

Whenever we add, 2 positive numbers or 2 negative numbers, the resultant answer will be above the capacity of resultant register.

Ex: - 6 + 7

6	=	0	1 1 0	
7	=	0	1 1 1	
-5	=	1	1 0 1	(carry from magnitude is added to the sign bit).

So go for 8-bit register.

6	=	0	0 0 0 0 1 1 0
7	=	0	0 0 0 0 1 1 1
13	=	0	0 0 0 1 1 0 1

- so, resultant register should be high enough to hold the result including its sign.
- This carry from the magnitude bit into sign bit is called as "overflow".
- To overcome, overflow situation, we need to take the values according to registers capacity.

Variables:

Variable names are the names given to memory location. A variable is a location in the memory that can hold a value. An entity that may change during execution of a program is called a variable. These locations contain integer/character constants.

Ex: - int n;
 ↙ ↘
Data type variable

RULES FOR DECLARING VARIABLES:

1. A variable name is any combination of alphabets, digits or underscore
2. First character in the variable name must be an alphabet or underscore.
3. A Variable name should not be a keyword.
4. White spaces are not allowed.
5. No other characters except underscore (_) should be used.

Ex:-

int	n	;	allowed
float	area	;	allowed
int	area of circle;		not allowed
int	area_of_circle;		allowed
int	char	;	not allowed
float	int	;	not allowed

Keywords:

In 'C' every word is classified into either a keyword or an identifier(refer name of variable). All keywords have fixed meaning and this cannot be changed.

Keywords are reserve words whose meaning has been explained to the system.(i.e., C compiler). System defined words can be called as 'keywords'. Keywords cannot be used as variable names.

- Keywords serve as a basic building block for program statement.
- All the keywords must be written in lower case letters.
- There are 32 keywords available in 'C' language.

They are:

auto	else	long	switch
break	enum	register	typedef
case	extern	return	union

char	float	short	unsigned
const	for	signed	void
continue	goto	sizeof	volatile
default	if	static	while
double	int	struct	do

- main() is not a keyword.

Arithmetics in C:

'c' evaluates arithmetic expressions in a sequence determined by the following rules of precedence:-

- Expressions in pair of parenthesis are evaluated first. If there are more than one pair of parenthesis, then, innermost parenthesis expressions are evaluated first.

Ex:

(10 (5 (6 + 2)))

└───────────┴───────────┘ This expression is evaluated first

- Multiplication, division and modulus operations are evaluated next. If an expression contain several multiplication, division and modulus operations , then evaluation starts from left and then proceeds to right.
- Addition and subtraction operations are evaluated at last. If many + and – operators are there in an expression, then evaluation proceeds from left to right.

Example :-

Evaluate the expression $2*((9/3)+4*(5-2))$

= 2 * ((9/3) + 4 * (5-2))

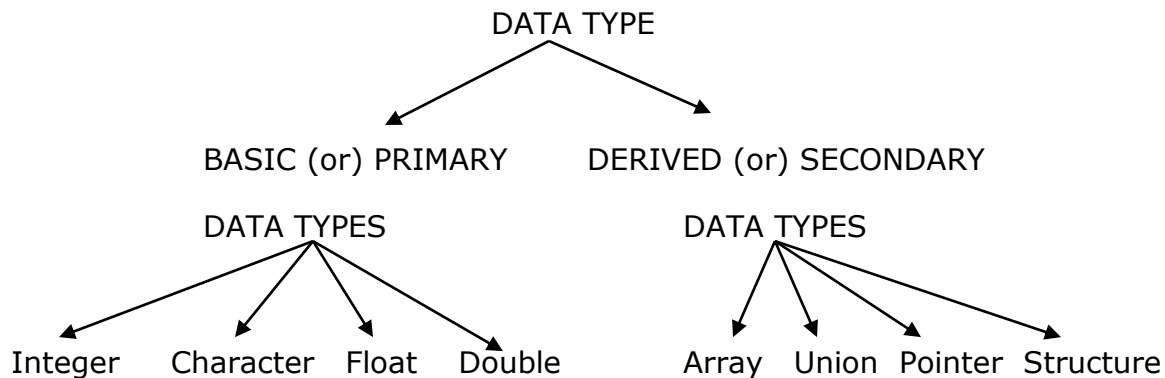
= 2 * (3 + 4 * (5-2))

$$\begin{aligned}
 &= 2 * (3 + \underline{4 * 3}) \\
 &= 2 * (\underline{3} + \underline{12}) \\
 &= \underline{2} * \underline{15} \\
 &= 30
 \end{aligned}$$

MATHEMATICAL EXPRESSION	C - EQUALENT FORM
$a+b$	$a+b$
$\frac{x * y}{z}$	$(x * y) / z$
$\frac{(a+b)^2}{(a-b)^2}$	$((a+b) * (a+b)) / ((a-b) * (a-b))$
$2x^2 + 3y + 1$	$2*x*x + 3*y + 1$

DATA TYPES IN C

There are various kinds of data types available in c that allows programmer, to use appropriate data type for the program.



DATA TYPES	NUMBER OF BYTES	FORMAT SPECIFIER	RANGE
Signed Char	1	%c	-128 to 127
Unsigned Char	1	%c	0 to 255
Short Signed Int	2	%d	-32768 to 32767
Short Unsigned Int	2	%u	0 to 65535
Long Signed Int	4	%ld	-2147483648 to

			2147483647
Long Unsigned Int	4	%lu	0 to 4294967295
Float	4	%f	-3.4e38 to 3.4e38
Double	8	%lf	-1.7e308 to 1.7e308
Long Double	10	%Lf	-1.7e4932 to 1.7e4932

TO PRINT TEXT IN C :

- In order to print text or values in c-language we printf() statement.
- This printf() statement is defined in a header file called stdio.h (standard input output)
- **Syntax for printf() statement is :**
printf(“ text to be displayed”);
- In order to print the value of a variable printf() should be written in the following way: **printf(“ format specifier ” , variable);**

TO ASSIGN VALUES TO VARIABLES IN C :

Two ways of assigning values to variables in the program is:

1. Initializing the value in the program itself.

Ex: int x = 20;

2. Assigning values at run-time (using scanf() statement)

SYNTAX OF SCANF STATEMENT :

Scanf() statement is used to assign values at run-time i.e., at the time of executing the program.

Syntax for scanf() is :

scanf (“format specifier “, address of variable);

example:

```
int x;  
scanf("%d",&x);  
printf("%d", x);
```

EVALUATION OF EXPRESSIONS:

An arithmetic expression is a combination of variables, constants and operators arranged as per the syntax of the language. Every arithmetic expression is evaluated according to the precedence rules. Precedence rules tells to compiler that while executing expressions, what operators are executed first, what are their executions and what is result.

The basic evaluation procedure includes two or more left-to-right passes through the expression. During the first phase the higher priority operators (if any) are evaluated. In next phase the next level operators are applied. This continues upto last level.

The order of precedence for different operators is :

<u>Precedence</u>	<u>Execution Order</u>
(), [], ->	left-to-right
++, --	right-to-left
!	right-to-left
Unary minus(-)	right-to-left
Sizeof	right-to-left
*, /, %	left-to-right
+, -	left-to-right
<<, >>	left-to-right
<, <=, >, >=	left-to-right
==, !=	left-to-right
&&	left-to-right
	left-to-right
?:	right-to-left
=, %=, *=, /=, +=, -=	left- to-right

OPERATORS IN C-LANGAUGE

OPERATOR: An operator is a symbol which instructs the computer to perform a certain task. The operand is a value on which the operation is performed.

OPERAND : A variable or constant on which the operation is performed.

Ex: $a + b = c$ [here, a,b,c are known as operands and +,= are known as operators.]

An operator may either Unary or Binary.

Unary operator means which act on only single operand. Ex: $a++$

Binary operator means which acts on two operands. Ex: a+b

The types of operators present in c are:-

- | | |
|------------------------------------|--------------------------|
| 1. Arithmetic operators | 2. Assignment operators |
| 3. Bitwise operators | 4. Conditional operators |
| 5. Increment & Decrement operators | 6. Logical operators |
| 7. Relational operators | 8. Special operators |

1. **Arithmetic operators:**

The different arithmetic operators are:

+ (Addition), - (Subtraction), * (multiplication), / (Division) and % (Modulus)

Here + & - act as both Unary and Binary.

Any operation between two integers yields result as an Integer.

Any operation between two floats yields results as a float.

Any operation between an integer and float results in float value.

i.e. If two operands are of different type, then the result is larger data type among these two.

Difference between division and modulo division is that :

Consider:

$$445 / 10 = 44$$

$$440 \% 10 = 5$$

When we divide 445 with 10 its quotient will be 44 which is result or normal division operation.

Where as for modulo division the result of integer division will be the result of modulo division. When we divide 445 with 10 remainder is 5 which is result of modulo division.

2. **Relational operators:**

These are used for comparison. These are used for decision making.

An expression containing a relational operator is termed as a relational expression.

The value of a relational expression is either zero or one. If the value is zero then it is false, if the value is nonzero then it is true.

The different relational operators are:

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

A simple relational expression contains only one relational operator and takes the following form: **ae1 relational operator ae2**.

3. Logical operators:

&& (Logical AND), || (Logical OR), ! (Logical NOT)

&&, || are used when we want to test more than one condition and make decisions. An expression which combines two or more relational expressions is termed as a logical expression. Logical expressions yields zero or nonzero according to the truth table.

OP1	OP2	&&		OP	! NOT
Nonzero	Nonzero	Nonzero	Nonzero	Nonzero	Zero
Nonzero	Zero	Zero	Nonzero	Zero	Nonzero
Zero	Nonzero	Zero	Nonzero		
Zero	Zero	Zero	Zero		

4. Assignment operators:

These are used to assign the result of an expression to a variable.

The general form is: **V op = exp;**

Here V is a variable expression is an expression and op is a binary arithmetic operator.

V op = exp is equal to v = v op (exp);

Ex: a+=10; is equal to a=a+10;

These are

+=, -=, *= and %=

These are also called **shorthand operators**. The advantages are:

- The left side operand need not be repeated and therefore easier to write.
- The statement is more concise and easier to read.
- The statement is more efficient.

5.Increment and Decrement operators:

C has 2 very useful operators which are generally not found in other languages.

These are **++ (Increment)**, **-- (decrement)**

++integer_variable indicates **pre-increment operation** in which, first the value of variable is incremented and then specified instruction is performed.

integer_variable++ indicates **post-increment operation** in which, first the specified instruction is performed and then the value of variable is incremented .

--integer_variable indicates **pre-decrement operation** in which, first the value of variable is decremented and then specified instruction is performed.

integer_variable-- indicates **post-decrement operation** in which, first the specified instruction is performed and then the value of variable is decremented.

Ex:-

```
int  a=10 , b ;
```

Pre -increment

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

```
b = a ;
```

```
a = a + 1 ;
```

```
b = 10 , a = 11 ;
```

Post - increment

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

```
a = a + 1 ;
```

```
b = a ;
```

```
b = 11 , a = 11 ;
```

Pre - decrement

```
b = a-- ;
```

```
b = a ;
```

```
a = a - 1 ;
```

```
b = 10 , a = 9 ;
```

Post - decrement

```
b = --a ;
```

```
a = a - 1 ;
```

```
b = a ;
```

```
b = 9 , a = 9 ;
```

6. Conditional or Ternary operators:

A ternary operator pair “?:” is available in C to construct conditional expressions of the form

exp1?exp2:exp3;

Here exp1 is evaluated first. If it is nonzero (true), then the expression exp2 is evaluated and becomes the value of the expression. If exp1 is false, exp3 is evaluated and its value becomes the value of the expression. Only one of the expressions (either exp2 or exp3) is evaluated.

7. Bitwise operators:

C has a distinction of supporting special operators known as bitwise operators for manipulation of data at the bit level. These operators are used for testing the bits, or shifting them right or left. Bitwise operators may not be applied to float or double. These are & (bitwise AND), | (bitwise OR), ^ (bitwise ExOR), <<(shift left), >> (shift right), ~ (complement).

Bit1	Bit2	&		^
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

Bit	~
0	1
1	0

Examples:

a=5	b=11	
a&b	a b	a^b
00000101	00000101	00000101
00001011	00001011	00001011
-----	-----	-----
00000001	00001111	00001110

$\sim a = \sim(00000101)$

11111010

<<: Left Shift

Bits will be moved towards left by one position ,0 will be added at the right most bit and left most bit will be lost.

$a << 2$

$5 << 1 = 00001010$

$<< 2 = 00010100$

>> : Right Shift

Bits will be moved towards right by one position ,0 will be added at the left most bit and right most bit will be lost.

$a >> 2$

$5 >> 1 = 00000010$

$>> 2 = 00000001$

8. Special operators:

These are comma operator, sizeof() operator, Ampersand(&). Comma(,) operator can be used to link the related expressions together.

Sizeof operator returns the number of bytes required for the operand storage. Ampersand symbol is used to denote the address of a variable.

Example :

```
int x;  
  
printf("%d", sizeof( x ) ); // output will be 2
```

TYPE CONVERSION:

When an operator has operands of different types, they are converted to a common type according to a small number of rules. In general, the only automatic conversions are those that convert a ``narrower" operand into a ``wider" one without losing information, such as converting an integer into floating point in an expression like `f + i`. Expressions that don't make sense, like using a `float` as a subscript, are disallowed. Expressions that might lose information, like assigning a longer integer type to a shorter, or a floating-point type to an integer, may draw a warning, but they are not illegal.

If either operand is `long double`, convert the other to `long double`.

- Otherwise, if either operand is `double`, convert the other to `double`.
- Otherwise, if either operand is `float`, convert the other to `float`.
- Otherwise, convert `char` and `short` to `int`.
- Then, if either operand is `long`, convert the other to `long`.

explicit type conversions can be forced (``coerced") in any expression, with a unary operator called a `cast`. In the construction

(type name) expression;

the *expression* is converted to the named type by the conversion rules above. The precise meaning of a cast is as if the *expression* were assigned to a variable of the specified type.

EX:

```
int y;  
y = (int) 9.8;  
printf("%d", y); // output will be 9 that is fraction part .8 is truncated.
```

CONTROL STRUCTURES IN C-LANGUAGE:

Some times in a program it will be necessary to :-

- Select a set of statements from several alternatives
- Skip certain statements depending on some conditions in the program and continue execution from some other point
- Repeat the execution of certain statements depending on the truthness of the condition.

Under such conditions, the control statements are used. There are three types of control structures:-

- ➔ Sequence control structure
- ➔ Selection control structure
- ➔ Repetition control structure

SEQUENCE CONTROL STRUCTURE

In this structure, the statements are executed in the same order in which they appear in the program.

```
Ex:- #include<stdio.h>
      main()
      {
        printf("hello world");
        printf( " welcome to c");
      }
```

In the above program both the statements are executed in the order in which they are written.

SELECTION CONTROL STRUCTURE :

These statements are also called as CONDITIONAL CONTROL STATEMENTS.

In some situations, it is necessary to check the condition to make the decision.

This involves performing a **logical test**. This results in either **true** or **false**.

Depending on the truthness of condition ,some statements are executed.

Thus, this is known as **conditional execution**. C provides various kinds of control statements. They are as follows:-

- **if**-statement
- **if-else** statement
- **else-if** ladder
- **nested-if** statement
- **switch** statement

if statement:-

It is used to execute a statement or set of statements conditionally. It is a simple 'if' statement.

Syntax:-

```

if(condition)
{
    Statements;
}
Statement-x;

```

The logical condition will be tested. It results in either true or false. If the condition is true the statements are executed or else not executed and the control is transferred to the next executable statement i.e, statement-x.

The simple-if statement is demonstrated by a sample program given below:-

```

main()
{
    int a;
    printf(" enter value of a ");
    scanf("%d",&a);
    if( a>10)
    {
        printf(" a is greater than 10");
    }
    printf( " if statement not executed");
}

```

In the above program:-

- If the value of 'a' is greater than 10 then the statement in if block is executed.
- If the value of 'a' is less than 10 then the if block will not be executed and the control transfers to next executable statement i.e, printf(" if statement is not executed").

if-else statement:

The if/else structure performs an action if the condition is true and performs a different action if condition is false. The if/else structure is called **double-selection structure** because it selects between two different actions.

Syntax:

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

If the condition is true, then statement-1 is executed, other wise; the control moves to else part and statement-2 is executed.

If-else is demonstrated by the following program:-

/* program to find whether a number is even or odd */

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

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

```
void main()
```

```
{
```

```
    int a;
```

```
    printf("Enter the no\n");
```

```
    scanf("%d",&a);
```

```
    if(a%2==0)
```

```
    printf(" %d is even");
```

```
    else
```

```
}
```

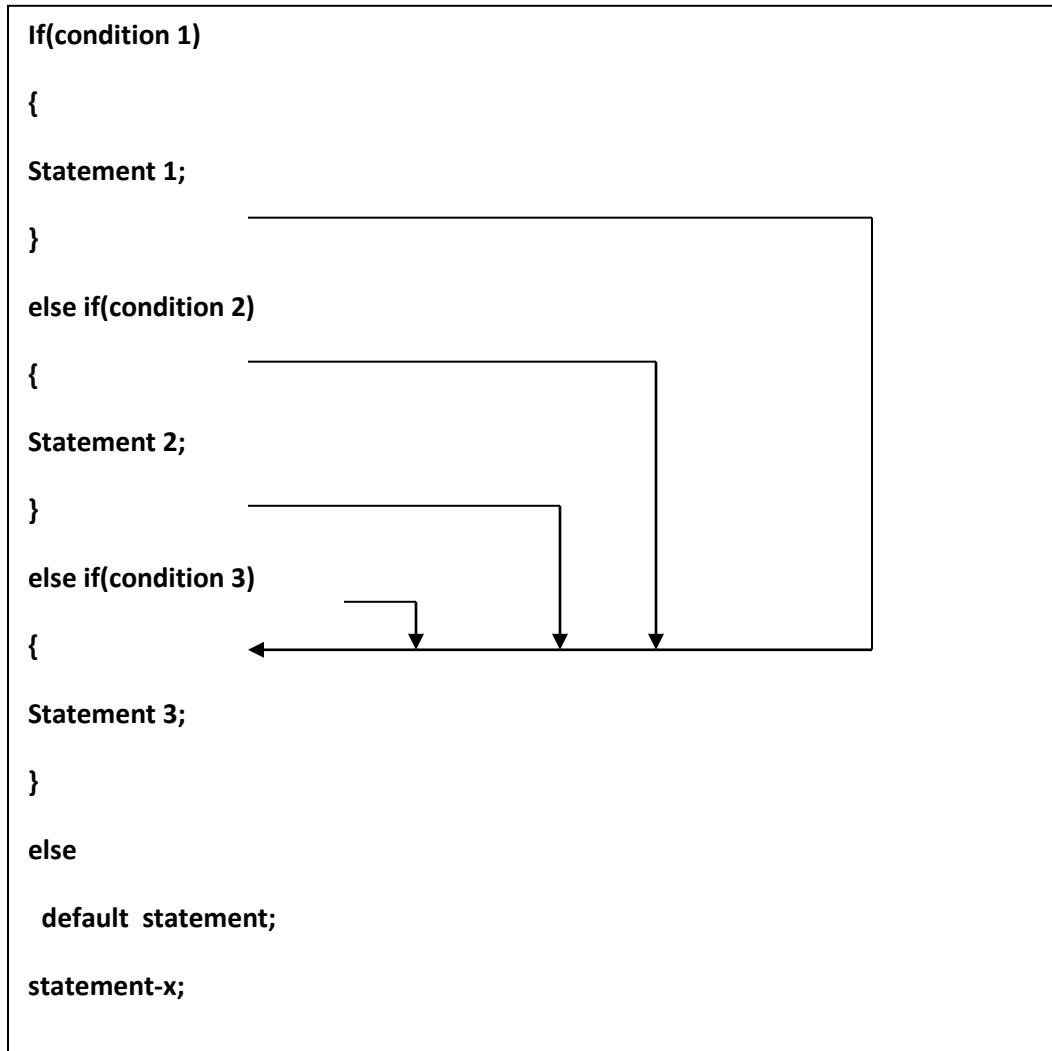
OUTPUT :Enter the no 21
21 is odd

Value of 'a' is given as 21...and 21%2 is not equal to zero...so, the control is transferred to else part leaving if block.

Else-if ladder :

There is another way of putting **ifs** together when multipath decisions are involved. A multipath decision is a chain of **ifs** in which the statement associated with **else** is an **if**

Syntax of else-if ladder is :-



Example:

/*program to calculate roots of quadratic equations*/

```
#include<stdio.h>
#include<math.h>
main()
{
    int a,b,c;
    float r1,r2,d;
    printf("Enter the coefficients of equation\n");
    scanf("%d %d %d",&a,&b,&c);
    d=(b*b)-(4*a*c);
    if(d<0)
    {
        printf("The roots are imaginary");
    }
    else if(d==0)
    {
        printf("Roots are equal\n");
        r1=-b/(2*a);
        r2=r1;
        printf("r1=%f  r2=%f",r1,r2);
    }
    else if(d>0)
    {
        printf("Roots are real");
        r1=(-b+sqrt(d))/(2*a);
        r2=(-b-sqrt(d))/(2*a);
        printf("r1=%f r2=%f",r1,r2);
    }
}
```

Nested –if statement :

Writing one “if” statement within another “if” statement is known as nested-if statement. When there are more than one condition to be tested then nested-if structure is used. The syntax for nested-if statement is as follows:-

```
if(condition-1) /* if condition-1 is true then condition-2 is tested*/
{
    if(condition-2) /* if condition-2 is true then statement-1 is executed*/
    {
        Statement-1;
    }
    else /* if condition-1 is true and condition-2 is false the statement-2 is
        executed */
        Statement-2;
}
else if(condition-3) /* if both condition1 and 2 are false then control comes
to else-if block and condition-3 will be tested.if it is
{
    true statement- 3 is o/p or else statement-4 is o/p*/
    Statement-3;
}
else
Statement-4;
```

Nested-if statement is demonstrated by a sample program:-

```
/*Program to find greatest of three no's */
```

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

```
main()
```

```
{
```

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

```
    printf("Enter the values\n");
```

```
    scanf("%d %d %d",&a,&b,&c);
```

```
    if(a>b)
```

```
    {
```

```
        if(a>c)
```

```
            printf("%d is greater",a);
```

```
        else
```

```
            printf("%d is greater",c);
```

```
    }
```

```
    else
```

```
    {
```

```
        if(b>c)
```

```
            printf(" %d is greater", b);
```

```
    }
```

```
    else
```

```
        printf(" %d is greater", c);
```

```
}
```

Output : Enter the values 10 54 40 54 is greater

Switch Statement :

The switch selection structure performs one of many different actions depending on the value of an expression. The switch structure is called **multiple-selection structure** because it selects one option among many different options available. The syntax for switch statement will be:-

```
switch(expression)
{
    case label1 : block-1;
                break;
    case label2 : block-2;
                break;
    case label3 : block-3;
                break;
    case label4 : block-4;
                break;
    .....
    .....
    case labeln : block-n;
                break;
    default : block;
            break;
}
Statement-x;
```

The expression is an integer expression or characters.

Each of the label values should be unique within a switch statement.

block-1, block-2... are statements lists and may contain zero or more statements. There is no need to put braces around these blocks. Note that case labels end with a **colon(:)**

When the switch is executed, the values of the expression is successively compared against the values label1, label2.... if a case is found whose value matches with the value of the expression, then the block of statements that follows the case are executed.

The **break** statement at the end of each block signals the end of a particular case and causes an exit from the **switch** statement, transferring the control to the statement-x following the switch.

The default is an optional case. When present, it will be executed if the value of the expression does not match with any of the case values. If not present, no action take place if all matches fail and the control goes to the statement-x.

Ex:-

/* Calculator program using switch case */

```
#include<stdio.h>
main()
{
    int a,b,c,n;
    printf("Enter two no's\n");
    scanf("%d %d", &a,&b);
    printf("1.Addition\n2.Substraction\n3.Multiplication\n4.Division");
    scanf("%d",&n);
    switch(n)
    {
        case 1:c=a+b; break;
        case 2:c=a-b; break;
        case 3:c=a*b; break;
        case 4:c=a/b; break;
        default: printf("Invalid Number\n");
    }
    printf("%d",c);
}
```

REPETITION CONTROL STRUCTURE :

C provides three repetition structures

- 1) while loop
- 2) do-while loop
- 3) for loop

A loop is a group of instructions the computer executes repeatedly while some loop-continuation condition remains true.

Repetition control structures are also called as **iterative control structures** or **looping control structures**.

WHILE LOOP

The basic format of while loop is

```
While(condition)
{
    body of loop
}
```

The while is entry-controlled loop statement. The condition is evaluated and if the condition is true then the body of the loop is executed. After execution of the body, the condition is once again evaluated and if it is true, the body is executed once again. This process of repeated execution of the body continues until the condition finally becomes false and the control is transferred out of the loop. On exit, the program continues with the statements. The braces needed only if the body contains one or more statements.

/* A simple program for while loop */

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

```
main()
```

```
{
```

```
    int a=1;
```

```

while (a>=3)
{
printf("a=%d",a);
a++;
}
printf("hello");
}

```

OUTPUT: hello

DO-WHILE LOOP

The general format of do-while is

<pre> do { body of loop } while(condition); </pre>
--

The while loop constructs that we have discussed in the previous section makes a test of condition before the loop is executed. Therefore, the body of the loop may not be executed at all if the condition is not satisfied at the very first attempt. On some occasions it might be necessary to execute the body of the loop before the test is performed. Such situations can be handled with the help of the **do** statement.

On reaching the do statement, the program proceeds to evaluates the body of the loop first. At the end of the loop, the condition in the while is evaluated. If the condition is true, the program continues to evaluates the body of the loop

once again. This process continues as long as the condition is true. When the condition become false, the loop will be terminated and the control goes to the statement that appears immediately after the while condition.

/* A simple program for do while loop*/

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

```
main()
```

```
{
```

```
    int a=1;
```

```
do
```

```
{
```

```
printf("a=%d\n",a);
```

```
a++;
```

```
} while (a>=3);
```

```
printf("Hello");
```

```
}
```

OUTPUT: 1

Hello

FOR LOOP

The for loop is another entry-controlled loop that provides a more concise loop control structure.

for(initialization; test-condition; increment)

```
{
```

```
    Statements;
```

```
}
```

The execution of the for statement is as follows:

- Initialization of the control variables is done first, using assignment statements such as `i=1` and `count=0`. The variable `i` and `count` are known as loop-control variables.
- The value of the control variable is tested using the test-condition. The test-condition is a relational expression, such as `i<10` that determines when the loop will exit. If the condition is true, the body of the loop is executed; otherwise the loop is terminated and the execution continues with the statement that immediately followed the loop.
- When the body of the loop is executed, the control is transferred back to the for statement after evaluating the last statement in the loop. Now, the control variable is incremented using an assignment statement such as `i=i+1` and the new value of the control variable is again tested to see whether it satisfied the loop condition. If the condition is satisfied, the body of the loop is again executed. This process continues still the value of the control variable fails to satisfy the test-condition.

Example:

```
/* program to demonstrate a simple for loop */
```

```
#include<stdio.h>

main()
{
    int a;
    for(a=1;a<=10;a++)
    {
        printf("%d\t",a);
    }
    getch();
}
```

OUTPUT: 1 2 3 4 5 6 7 8 9 10

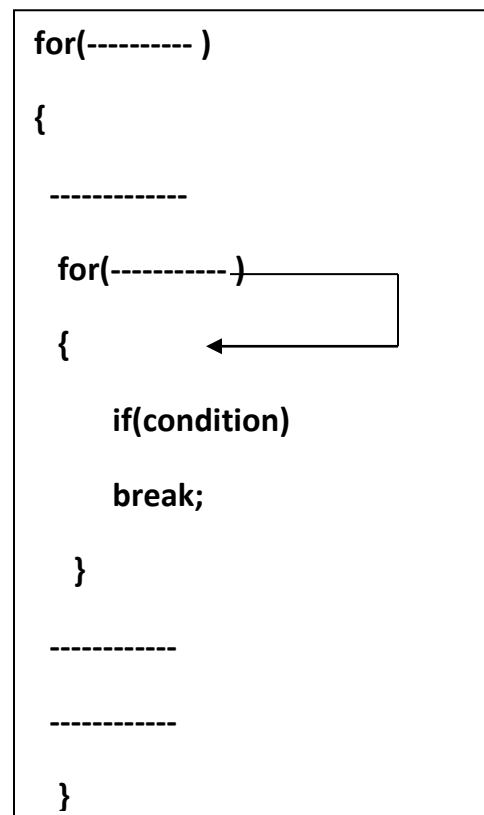
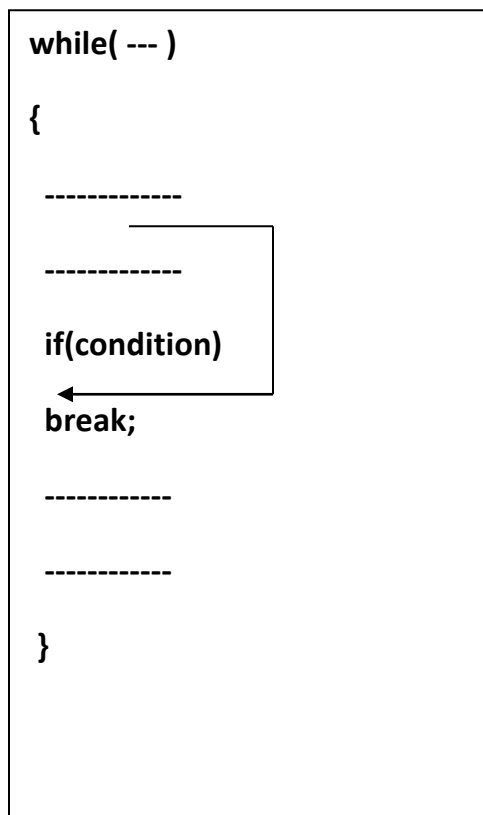
UNCONDITIONAL CONTROL STATEMENTS :

Without testing any condition the flow of program will be altered. There are 3 such statements in c :

1. break
2. continue
3. goto

BREAK statement :

When the break statement is encountered inside a loop, the loop is immediately exited and the program continues with the statement immediately following the loop. When the loops are nested, the **break** would only exit from the loop containing it. That is, break will exit only a single loop.



Example:

```
#include<stdio.h>

main()
{
    int i;
    clrscr();
    for(i=1;i<=10;i++)
    {
        if(i==5)
            break;
        printf("%d\t",i);
    }
    printf("\nBroke out of the loop at x == %d\n",x);
}
```

OUTPUT:

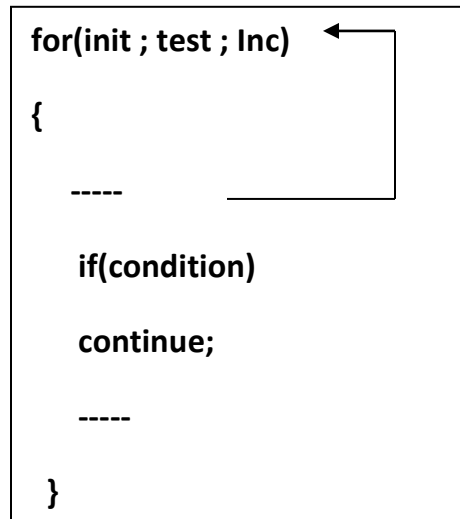
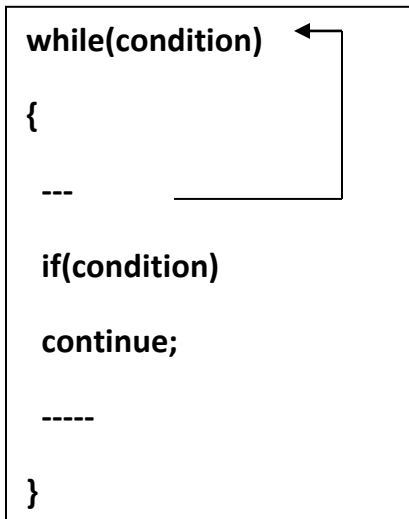
```
1 2 3 4
Broke out of the loop at x==5
```

CONTINUE statemen :

When the continue statement is encountered in the loop, as the name implies, it causes the loop to be continued with the next iteration after skipping any statements in between. The continue statement tells the compiler, "**SKIP THE FOLLOWING STATEMENTS AND CONTINUE THE NEXT ITERATION**". The format of the continue statement is

continue;

In **while** and **do** loops, continue causes the control to go directly to the test-condition and then to continue the iteration process. In the case of **for** loop, the increment section of the loop is executed before the test-condition is evaluated.



Example:

```

#include<stdio.h>
main()
{
    int i;
    for(i=1;i<=10;i++)
    {
        if(i==5)
            continue;
        printf("%d\t",i);
    }
    printf("\nUsed continue to skip printing the value 5",x);
}

```

OUTPUT:

```

1 2 3 4 6 7 8 9 10
Used continue to skip printing the value 5

```

GOTO :

It is used to alter the normal flow of control of a program. Goto requires a label in order to identify the place where the branch is to be made. A label is any valid variable name and must be followed by colon. The label is to be placed immediately before the statement where the control is to be transferred.

Syntax: **goto label;** (for goto)

label:

{

block of statements; (for label)

}

ex:

```
void main()
```

```
{
```

```
    int a=20;
```

```
    goto abc; /* here 'abc' is the name of the label */
```

```
    printf("going to goto statement");/*this statement is not executed*/
```

```
    abc: printf("%d",a); /*when goto statement executes control comes here*/
```

```
}
```

Output: 20

STORAGE CLASSES:

All variables will have not only data type, but also a storage class.

In order to define a variable totally, we need to mention both its data type and storage class.

Storage Class of a Variable defines:

- i) Where the variable is stored
- ii) Default initial value of variable.
- iii) Scope of variable i.e, in which functions the variable is available.
- iv) What is a life of a variable i.e, how long will the variable exist.

There are 4 types of storage classes in C:

1. Automatic storage classes
2. Register storage classes
3. Static storage classes
4. External storage classes

Inside **CPU**, 2 types of storage spaces are present

- i) memory unit.
- ii) registers. (limited in number)

a) Automatic storage classes:

Features of a variable having an automatic storage class will be:

Storage	→ memory
Default initial value	→ garbage value
Scope	→ Local to the block in which it is defined.
Life	→ Till the control remains within the block, the variable is defined.
Keyword	→ auto

```
include<stdio.h>void
main()
{
auto int i;
printf(“%d”,i);
}
```

Output: 1011 (unpredictable/unexpected value if differs from compiler to compiler)

Program to demonstrate scope and life of automatic variable:

```
main()
{
    auto int i;
    {
        {
            printf("%d",i);
        }
        printf("%d",i);
    }
}
```

Output: 1 1 1

Reason: in the above program, i is a automatic variable, whose scope is local, to the block, in which it is defined.

There fore when the control comes out of the block, in which the variable is defined. The variable and its value are lost.

```
main()
{
    auto int i=1;
    {
        auto int i=2;
        {
            auto int i=3;
            printf("%d",i);
        }
        printf("%d",i);
    }
    printf("%d",i);
}
```

Output:

3 2 1

Reason: Compiler treats 3 i's as totally different variables, since they are defined in different blocks. Once control comes out of the inner most block, the variable i with value 3 is lost. Therefore, i in 2nd printf() statement. Refers to i with value 2. When control comes out of the next

inner most blocks i=2 is lost and 3rd printf() statement. Refers to i with value i.

b) Register Storage class:

Features of a variable defined under register storage class are:

Storage	→ CPU register
Default initial value	→ garbage value
Scope	→ Local to the block in which variable is defined.
Life	→ Till the control remains within the block, the variable is defined.
Keyword	→ register

A value stored in CPU register, can be accessed faster than a value stored in memory. If a variable is to be used at many places/ many times in the program, then it would be better to declare its storage class as register.

Disadvantage: CPU registers will be limited in number. If all the register of CPU are busy with some other task, then variable storage class is taken as auto.

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

Output: 1 2 3 4 5 6 7 8 9 10

c) Static Storage Class:

Features of a variable defined with static storage class are:

Storage	→ Memory
Default initial value	→ Zero
Scope	→ Local to the block in which variable is defined.
Life	→ Till the control remains within the block, the variable is defined.
Keyword	→ static

Difference between static and automatic variables is that they don't disappear, when the function is not active. Their value exists and when control comes back to same function again, the static variables have the same values, they had last time.

Ex:

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

Output: 1 2 3

Reason: In the above program, if we declare variable i as integer and static variable the output will be 1 2 3.

Ex:

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

Output: 1 1 1

Reason: In the above program, when I is of automatic storage class, each time increment() is called, it will be re-initialized to 1. When function terminates value i=2 will be lost.

Instead of 'auto' if we use 'static', i will be initialized to 1 only once. During first call of increment 'i' will be 2 and i is of static type, the value will not be lost. Similarly during second function call of increment(), i will be 3.

d) External Storage Class:

Features of a variable defined with External storage class are:

Storage	→ Memory
Default initial value	→ Zero
Scope	→ Global
Life	→ As long as execution of program doesn't come to an end.
Keyword	→ Extern

External variables are declared outside the function, So they are available to all the functions in program.

Ex:

```
#include<stdio.h>
int i=20;
main()
{
extern int j;
printf("%d",i);
printf("%d",j);
}
int j=40;
```

Out put: 20 40

Keyword Extern indicates that variable j is defined some where after or outside the main().

Here i and j are global variable. Therefore they are defined outside function, both enjoy external storage class.

Difference between them is:

extern int j; → Declaration; initially memory is not registered for it.

Int j=40; → Defination

S.No	Storage Classes	Keyword	Storage	Default Value	Scope	Life Time
1	Automatic	Auto	CPU	Garbage	Local	→ Till the control remains within the block, the variable is defined.
2	Register	Register	Register	Garbage	Local	→ Till the control remains within the block, the variable is defined.
3	Static	Static	CPU	Zero	Local	Persist between diff function calls
4	External	Extern	CPU	Zero	Global	The variable exists through out the execution of the program.

ARRAYS

INTRODUCTION :

If a program to store the roll numbers of students of a class is to be written, then we will have 2 options:-

Suppose , students = 60

Option 1:- declare 60 different variables and store each roll no in a variable.

Option 2:- declare a single variable in such a way that, we can store all the roll no's in it.

Definitely, the programmer's choice will be option-2. In order to declare a variable, to store many values in it, we have to use the concept of "ARRAY" .

DEFINITION:

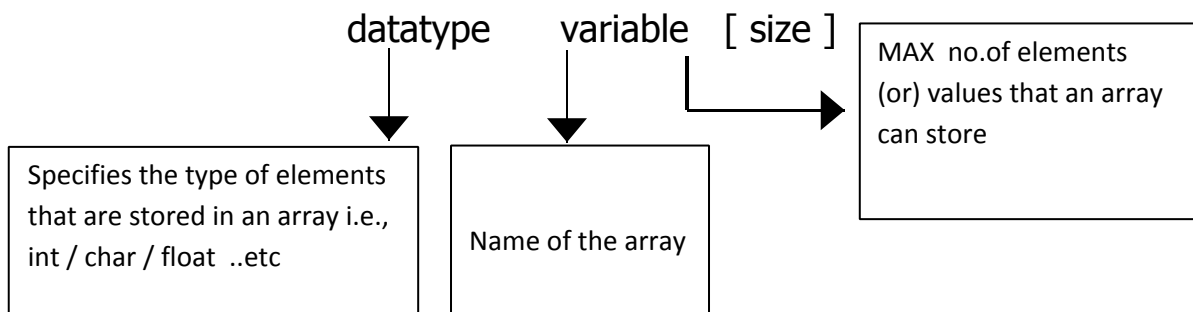
An array can be defined as, a group of related data items that share a common name. these data items may be int / char / float / double ..etc

- All these are stored in continuous memory locations (on RAM)
- Sometimes an array can be called as a SUBSCRIPTED VARIABLE

Difference between ARRAY and VARIABLE:

A variable can store only a single value at a time. Where as an array can store a multiple values at a time.

Syntax of Array:



Note : An array at a time can hold multiple values of similar data type only i.e., at a time array can hold group of integers, (or) group of floating point numbers etc.,

Declaration of variable:

```
int a;
```

this syntax indicates:-

- can hold 1 integer value
- only 2 bytes of memory is registered. Therefore it is integer data type.

Declaration of array:

```
int a [ 5 ] ;
```

- can hold 5 integer values
- for each integer value 2 bytes of memory will be registered i.e.,

$5 * 2 = 10$ bytes of memory is registered

Total memory size of Array:

Total size = size * (size of data type)

Ex:-

(i) float k[10];

total size = $10 * 4 \Rightarrow 40$ bytes

(ii) char c[5];

total size = $5 * 1 \Rightarrow 5$ bytes

Declaring an array and Defining an array:

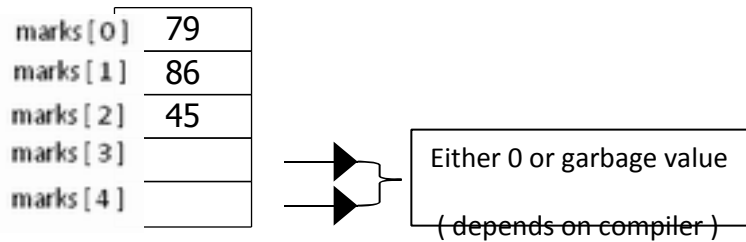
(i) int marks [5] = { 70, 86, 45, 67, 89 };

in the above declaration, the array 'marks [5]' will store the values as shown below:

marks [0]	79
marks [1]	86
marks [2]	45
marks [3]	67
marks [4]	89

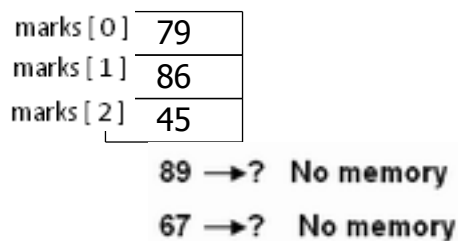
(ii) int marks [5] = { 79, 86, 45 };

in the above declaration the values are stored in memory as follows:-



(iii) `int marks [3] = {79, 86, 45, 89, 67};`

Memory allocation will not be done properly for the above declaration and it results in an error - (too many parameters in declaration)



This is because , size of array is given as only 3. Therefore for 4th and 5th elements memory will not be allocated, which results in an error.

(iv) `int marks [3] = { 79, 86, 45, 89, 67 } ;`

for this declaration , memory will be allocated based on the input values of an array.
i.e.,

in above declaration , 5 values are being given as inputs, the compiler automatically, fits the size of array as 5 and gives 5*2 bytes = 10 bytes of memory.

NOTE: Initialization of an array must be done using curly braces (flower brackets) only.

Datatype name[size] = {element1,...};

WITHOUT ARRAY

```
void main()
{
int a=10, b=20, c=30;
printf(" a = %d ,", a );
printf("b = %d ,", b );
printf("c = %d .", c );
}
output:
a=10 , b=20 , c=30 .
```

WITH ARRAY

```
void main()
{
int a[5]= {10, 20, 30}, i;
printf("array elements are:");
for(i=0;i<5;i++)
{
printf(" %d ",a[i]);
}
}
Output: array elements are: 10 20 30
```

NOTE: all the array elements are numbered, starting from zero (0).

```
int m[5] = {75, 85, 68, 97, 54};
```

In above declaration,

m[2] = 68 i.e. 3rd element of the list, as it starts from 0, not 1.

ENTERING DATA INTO AN ARRAY:

Generally, values at runtime can be assigned to an array using "for loop".

Ex: printf("enter marks");

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

```
{
```

```
scanf("%d",&marks[i]);
```

```
}
```

The for loop causes the process of assigning values at a runtime, till i=9 i.e. 10 values (0-9) are stored in an array. Values will be stored, in the array as:-

Marks[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]

// WAP to accept values for five integers, at runtime and display them.

```
Void main()
```

```
{
```

```
Int a[20], i;
```

```
Printf("enter values into array");
```

```
For(i=0;i<5;i++)
```

```
{
```

```
Scanf("%d",&a[i]);
```

```
}
```

➔ To accept values into an array.

```
Printf("values in the array are:");
```

```
For(i=0;i<5;i++)
```

```
{
```

```
printf("%d",&a[i]);
```

```
}
```

➔ To display values of array.

NOTE: Even to print the values of array, “for loop” should be used

//Program to accept 2 arrays and add the corresponding elements of those arrays

```
void main()
{
    int a[10], b[10], sum[10];
    int i, j;
    printf("enter size of Arrays");
    scanf("%d", &j);
    printf("enter values for array – 1");
    for (i=0; i<j; i++)
    {
        scanf("%d",&a[i]);
    }
    printf("enter values for array – 2");
    for (i=0;i<j;i++)
    {
        scanf("%d",&b[i]);
    }
    for(i=0;i<j;i++)
    {
        sum[i]=a[i]+b[i];
    }
    printf("sum of arrays is : ");
    for(i=0;i<j;i++)
    {
        printf("%d",sum[i]);
    }
}
```

OUTPUT:-

```
Enter size of arrays 4
Enter values for array -1  10 20 30 40
Enter values for array -2  5 10 15 20
Sum of arrays is : 10 30 45 60
```

Elements are stored in array as shown below:-

a[0]	10		b[0]	5		sum[0]	15
a[1]	20		b[1]	10		sum[1]	30
a[2]	30	+	b[2]	15	=	sum[2]	45
a[3]	40		b[3]	20		sum[3]	60

//Program to print the largest element of the array

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

```
Main()
```

```
{
```

```
    int x[6], large;
```

```
    int i;
```

```
    printf("enter elements in array");
```

```
    for(i=0; i<6;i++)
```

```
    {
```

```
        scanf("%d",&x[i]);
```

```
    }
```

```
    large=x[0];                /*Assuming x[0] is large */
```

```
    for(i=1, i<6;i++)
```

```
    {
```

```
        if(x[i]>large)
```

```
        {
```

```
            large=x[i];
```

```
        }
```

```
    }
```

```
    printf("largest element is %d",large);
```

```
}
```

SEARCHING

Gathering any information (or) trying to find any data is said to be a Searching process.

Searching technique can be used more efficiently if the data is present in an ordered manner.

Most widely used Searching methods are:-

- i) Linear Search (Sequential Search)
- ii) Binary Search

LINEAR SEARCH:-

Suppose an array is given, which contains 'n' elements. If no other information is given and we are asked to search for an element in array, then we should compare that element, with all the elements present in the array. This method which is used to Search the element in the array is known as Linear Search. Since the key element/ the element which is to be searched in array, is found out by comparing with every element of array one-by-one, this method is also known as Sequential Search.

Example:-

An array 'x' is given, which contains 5 elements in it i.e.,

`int x[5] = { 75, 52, 61, 43, 88};`

These are stored in the memory in continuous memory location.

Fig.(a)

75	52	61	43	88
x[0]	x[1]	x[2]	x[3]	x[4]

We are asked to search for an element '43' in the array.

Then we have to compare '43' with each and every element of the array. This is represented in the below figures:-

Fig.(b)

75	52	61	43	88
↑				
43				

75	52	61	43	88
		↑		
		43		

Fig.(d)

fig.(c)

75	52	61	43	88
			↑	
			43	

75	52	61	43	88
			↑	
			43	

fig.(e)

// WAP to demonstrate LINEAR SEARCH

```
#include<stdio.h>
#include<conio.h>
Void main()
{
int linear[20],n,i,k,temp=0;
clrscr();
printf(" enter range of elements");
scanf(" %d",&n);
printf("enter elements into array");
for(i=0; i<n; i++)
{
scanf("%d",&linear[i]);
}
printf("enter search key:");
scanf("%d",&k);

for(i=0, i<n; i++)
{
if(k==linear[i])
{
temp=1;
printf("%d is found at location %d", k, i+1);
break;
}
}
if(temp!=1)
{
printf("element not found");
}
getch();
}
```

OUTPUT:-

```
Enter range of elements
5
Enter elements into array
45 68 75 83 99
Enter search key
83
83 is found at location 4
```

Reason for using 'temp' variable:

Without "temp", if we write else statement (or) else-if, they should be written after 'if' inside for loop. If we write outside for loop and error misplaced else will occur.

If we write inside for loop, every time those statements will be executed. Therefore we use 'temp' variable and assign its value to 1, and use the 'temp' variable in 'if' condition. Whenever search key is not found in the list.

EXPLANATION:-

Number of elements for which Linear search technique is to be performed, should be taken i.e. array size. (n).

Let n=5.

5 elements are entered into an array using for loop and stored in continuous memory locations.

	1	2	3	4	5
	72	45	68	59	83
Linear	[0]	[1]	[2]	[3]	[4]

The element which is to be searched is taken i.e., Search key element (k)

k=59

According to the linear search method, now, '59' should be compared with every element in the list and when '59' matches with the element in the list, then its position should be displayed. To implement this logic, the c-Program code is:-

```
for(i=0; i<n; i++)
{
    if(k==liner[i]);
    {
        temp=1;
        printf("%d found at %d",k, i+1);
    }
}
if (temp!=1)
printf("element not found");
```

BINARY SEARCH:-

Efficient search method for large arrays. Linear search, will required to compare the key element, with every element in array. If the array size is large, linear search requires more time for execution.

In such cases, binary search technique can be used.

To perform binary search:-

- i) Elements should be entered into array in Ascending Order
- ii) Middle element of the array must be found. This is done as follows

Find the lowest position & highest position of the array i.e., if an array contains 'n' elements then:-

Low=0

High =n-1

Mid =(low+high)/2

Note: We are calculating mid position of the array not the middle element. The element present in the mid position is considered as middle element of array.

Now the search key element is compared with middle element of array. Three cases arises

Case 1 : If middle element is equal to key, then search is end.

Case 2 : If middle element is greater than key, then search is done, before the middle element of array.

Case 3 : If middle element is less than key, then search is done after the middle element of array.

This process is repeated till we get the key element (or) till the search comes to an end, since key element is not in the list.

//PROGRAM TO DEMONSTRATE BINARY SEARCH

```
#include<stdio.h>
#include<conio.h>
Void main()
{
int binary[20],n, i, k, low, mid, high;
clrscr();
printf("enter range of elements");
scanf("%d",&n);
printf("enter elements into array");
for(i=0, i<n; i++)
{
scanf("%d",&binary[i]);
}
printf("enter search key");
```

```

scanf("%d",&k);
low=0;
high=n-1;
while(low<=high)
{
    mid=(low+high)/2;
    if(binary[mid]<k)
    {
        low=mid+1;
    }
    else if(binary[mid]>k)
    {
        high=mid-1;
    }
    else
        break;
}
if(binary[mid]==k)
{
    printf("element is found at location %d",mid+1);
}
else
    printf("element not found");
getch();
}

```

OUTPUT:-

```

Enter the range of elements
5
Enter elements into array
22  28  34  45  58
Enter search key
34
Element found at location 3

```

SORTING

Sorting means arranging the given data in a particular order.

Some of the Sorting techniques are:

- i) Bubble sort
- ii) Selection sort

BUBBLE SORT

In bubble sort each element is compared with its adjacent element.

If first element is larger than second one, then both elements are swapped. Other wise, element are not swapped.

Consider the following list of numbers.

Example:

74	39	35	97	84
a[0]	a[1]	a[2]	a[3]	a[4]

74 & 39 are compared

Bcoz $74 > 39$, both are swapped. Now list is

39	74	35	97	84
a[0]	a[1]	a[2]	a[3]	a[4]

74 & 35 are compared

Bcoz $74 > 35$, both are swapped, Now list is

39	35	74	97	84
a[0]	a[1]	a[2]	a[3]	a[4]

74 & 97 are compared

Bcoz $74 < 97$ list remain unchanged

97 & 84 are compared

Bcoz $97 > 84$, both are swapped, the list becomes

39	35	74	84	97
a[0]	a[1]	a[2]	a[3]	a[4]

Note: After first pass, largest element in given list occupies the last position.

After second pass, second largest element is placed at second last position and so on..

//PROGRAM TO DEMONSTRATE BUBBLE SORT

```

#include<stdio.h>
#include<conio.h>
void main()
{
    int i, j, n, bubble[20], temp;
    clrscr();
    printf("enter range of elements");
    scanf("%d",&n);
    printf("enter elements");
    for(i=0, i<n; i++)
    {
        scanf("%d",&bubble[i]);
    }
    for(i=0; i<n; i++)
    {
        for(j=0;j<n-1;j++)
        {
            if(bubble[j] > bubble[j+1])
            {
                temp=bubble[j];
                bubble[j]=bubble[j+1];
                bubble[j+1]=temp;
            }
        }
    }
    printf("after sorting");
    for(i=0; i<n; i++)
    {
        printf("%d",bubble[i]);
    }
}

```

OUTPUT

enter range of elements

5

Enter elements

3 2 5 4 6

After sorting

2 3 4 5 6

SELECTION SORT:

In selection sort, element at first location (0th location) is considered as least element, and it is compared with the other elements of the array. If any element is found to be minimum than the element in first location, then that location is taken as minimum and element in that location will be the minimum element.

After completing a set of comparisons, the minimum element is swapped with the element in first location (0th location).

Then again element second location is considered as minimum and it is compared with the other elements of array and the process continues till the array is sorted in ascending order.

Note: After first pass, smallest element in given list occupies the first position.

After second pass, second largest element is placed at second position and so on..

//PROGRAM TO DEMONSTRATE SELECTION SORT

```
#include<stdio.h>
#include<conio.h>
void main()
{
    int i, j, n, a[100], t, min;
    clrscr();
    printf("enter range of elements");
    scanf("%d",&n);
    printf("enter elements:");
    for(i=0, i<n; i++)
    {
        scanf("%d",&a[i]);
    }
    printf("elements before sorting");
    for(i=0; i<n; i++)
    {
        printf("%d",a[i]);
    }
    for(i=0; i<n-1; i++)
    {
        min=i;
        for(j=i+1; j<n; j++)
        {
            if(a[j]<a[min])
                min=j;
        }
        if(min!=i)
        {
            t=a[i];
            a[i]=a[min];
            a[min]=t;
        }
    }
}
```

```

    }
    printf("elements after sorting are");
    for(i=0; i<n; i++)
    {
        printf("%d", a[i]);
    }
    getch();
}

```

OUTPUT

enter range of elements

5

Enter elements

3 2 5 4 6

After sorting 2 3 4 5 6

TWO DIMENSIONAL ARRAYS

There are 3 types of arrays:-

1. 1 D Array
 2. 2 D Array
 3. Multi - Dimensional Array
- Maximum limit of Arrays is compiler dependent.

If we want to represent an array in a matrix form we will be using 2 D Arrays.

General form of an 2D array is

Datatype array_name[row_size][column_size];

Ex: `int i[4][3];`

An array 'i' is declared which contains 12 integer values in 4 rows and 3 columns.

Initializing a 2D array in program:

```
int    i[4][3] = { { 1,2,3 } , { 4,5,6 } , { 7,8,9 } , { 10,11,12 } };
```

or

```
int    i[4][3] = { 1,2, 3,4, 5, 6 , 7, 8, 9,10,11,12 };
```

or

```
int    i[][3] = { { 1,2,3 } , { 4,5,6 } , { 7,8,9 } , { 10,11,12 } };
```

NOTE: It is important to remember that while initialising an array it is necessary to mention the second(column) dimension, whereas the first dimension(row) is optional

$$\begin{matrix} & c1 & c2 & c3 \\ r1 & \left(\begin{array}{ccc} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \\ 10 & 11 & 12 \end{array} \right) & & 4 \times 3 \end{matrix}$$

MEMORY OF 2 D ARRAY:

In memory it is not possible to store elements in form of rows and columns. Whether it is a 1 D (or) 2 D Array, the elements are stored in continuous memory locations. The arrangement of elements of a 2 D is shown below:

```
int a[4][3] = { { 10,20,30 } , { 4,8,9 } , { 23,41,32 } , { 15,18,24 } };
```

```
a[0][0] a[0][1] a[0][2]  a[1][0]  a[1][1] a[1][2] .....
          .....a[3][2]
```

10	20	30	4	8	9	23	41	32	15	18	24
1000 1002		1004		1006		1008				
1022											

// WAP to read a 2-D Array and print it.

Void main()

{

int a[10][10] , i , j , m , n ;

Printf "enter the order of matrix \n ");

Scanf(" %d%d " , &m, &n);

Printf " \n enter the elements of array: \n ");

for(i=0;i<m;i++) // for rows.

{

for (j=0;j<n;j++) // for columns.

{

Scanf("%d", &a[i][j]);

}

}

for(i=0;i<m;i++) // for rows.

{

for (j=0;j<n;j++) // for columns.

{ printf("%d", a[i][j]);

}

print(" \n ");

} }

// Write a program for ADDITION OF 2 MATRICES

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

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

```
void main()
```

```
{
```

```
    int a[10][10],b[10][10],c[10][10];
```

```
    int i,j;
```

```
    int r1,r2,c1,c2;
```

```
    printf("Enter the size of the first matrix ( rows and coloums)\n");
```

```
    scanf("%d%d",&r1,&c1);
```

```
    printf("Enter the size of the seond matrix (rows and coloums)\n");
```

```
    scanf("%d%d",&r2,&c2);
```

```
    printf("Enter the elements in matrix one");
```

```
    for(i=0;i<r1;i++)
```

```
    {    for(j=0;j<c1;j++)
```

```
    {
```

```
        scanf("%d",&a[i][j]);
```

```
    }
```

```
}
```

```

printf("Enter the elements in matrix two");
for(i=0;i<r2;i++)
{
for(j=0;j<c2;j++)
{
scanf("%d",&b[i][j]);
} }
for(i=0;i<r1;i++)
{
for(j=0;j<c1;j++)
{
c[i][j]=a[i][j]+b[i][j];
}
}

printf("The sum of the two matrices is \n");
for(i=0;i<r1;i++)
{
for(j=0;j<c2;j++)
{

```

```
printf("  %d",c[i][j]);  
}  
printf("\n");  
}  
}
```

// Program for MULTIPLICATION OF 2 MATRICES

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

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

```
void main()
```

```
{
```

```
    int a[10][10],b[10][10],c[10][10],r1,c1,r2,c2,i,j,k;
```

```
    printf("Enter the number of rows and columns in 1st matrix\n");
```

```
    scanf("%d%d",&r1,&c1);
```

```
    printf("Enter the number of rows and columns in 2nd matrix\n");
```

```
    scanf("%d%d",&r2,&c2);
```

```
    if(c1 == r2)
```

```
    {    printf("Enter the elements of the matrix of a\n");
```

```
        for(i=0;i<r1;i++)
```

```
        {
```

```
            for(j=0;j<c1;j++)
```

```
            {
```

```
                scanf("%d",&a[i][j]);
```

```
            }
```

```
        }
```

```

printf("Enter the elements of the matrix of b\n");
for(i=0;i<r2;i++)
{
    for(j=0;j<c2;j++)
    {
        scanf("%d",&b[i][j]);
    }
}
for(i=0;i<r1;i++)
{
    for(j=0;j<c2;j++)
    {
        c[i][j]=0;
        for(k=0;k<c2;k++)
        {
            c[i][j] =c[i][j] + (a[i][k] * b[k][j]);
        }
    }
}

```

```

printf("\nProduct of the two matrices is\n");
for(i=0;i<r1;i++)
{
    for(j=0;j<c2;j++)
    {
        printf(" %d",c[i][j]);
    }
    printf("\n");
}
}
else
    printf("Matrix multiplication not possible");
}

```

// Program to find transpose of given matrix and print the identity matrix of order m x n

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

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

```
void main()
```

```
{
```

```
    int a[10][10],b[10][10],c[10][10],r1,c1,r2,c2,i,j,k;
```

```
    clrscr();
```

```
    printf("Enter the number of rows and columns\n");
```

```
    scanf("%d%d",&r1,&c1);
```

```
    printf("Enter the %d elements\n",r1*c1);
```

```
    for(i=0;i<r1;i++)
```

```
    {
```

```
        for(j=0;j<c1;j++)
```

```
        {
```

```
            scanf("%d",&a[i][j]);
```

```
        }
```

```
    }
```

```

for(i=0;i<r1;i++)
{
    for(j=0;j<c1;j++)
    {
        b[j][i] = a[i][j];
    }
}

printf("Transpose of given matrix is\n");
for(i=0;i<c1;i++)
{
    for(j=0;j<r1;j++)
    {
        printf("%d ",b[i][j]);
    }
    printf("\n");
}

if(r1 == c1)
{
    for(i=0;i<r1;i++)

```

```

        {
            for(j=0;j<c1;j++)
            {
                if(i==j)
                    a[i][j] = 1;
                else
                    a[i][j] = 0;
            }
        }

        printf("Identity matrix of given order is\n");
        for(i=0;i<r1;i++)
        {
            for(j=0;j<c1;j++)
            {
                printf("%d ",a[i][j]);
            }
            printf("\n");
        }
    }

    else

        printf("\nIdentity matrix should be square matrix");

}

```

// Program to check the EQUALITY OF 2 MATRICES

```
void main()
{
    int a[10][10],b[10][10];
    int sum,i,j,k,r1,r2,c1,c2,temp;
    printf("enter rows and columns matrix-1\n");
    scanf("%d %d",&r1,&c1);
    printf("enter rows and columns matrix-2\n");
    scanf("%d %d",&r2,&c2);
    if(r1==r2&& c1==c2)
    {   printf("enter elements of the matrix-1\n");
        for(i=0;i<r1;i++)
        {
            for(j=0;j<c1;j++)
            {
                scanf("%d",&a[i][j]);
                printf("\n");
            }
        }
    }
```

```

printf("enter elements of the matrix-2\n");
for(i=0;i<r2;i++)
{
    for(j=0;j<c2;j++)
    {
        scanf("%d",&b[i][j]);
        printf("\n");
    }
}

/*printing matrix-1*/
printf("matrix-1\n\n");
for(i=0;i<r1;i++)
{
    for(j=0;j<c1;j++)
    {
        printf("%d\t",a[i][j]);
    }
    printf("\n\n\n");
}

```

```

/* printing matrix-2*/
printf("matrix-2\n\n");
for(i=0;i<r2;i++)
{
    for(j=0;j<c2;j++)
    {
        printf("%d\t",b[i][j]);
    }
    printf("\n\n\n");
}

/* checking equality */
for(i=0;i<r1;i++)
{
    for(j=0;j<c2;j++)
    {
        if(a[i][j]!=b[i][j])
        {
            temp=1;
            break;
        }
    }
}

```

```
        }  
    }  
}
```

else

```
printf("cannot be compared");
```

```
if(temp==1)
```

```
printf("matrices are not equal");
```

else

```
printf("matrices are equal");
```

```
}
```

**// Program to check whether the given matrix is SYMMETRIC
MATRIX or not**

```
void main()
{
int a[10][10],b[10][10];

int sum,i,j,k,m,n,temp;

clrscr();

printf("enter size of square matrix\n");

scanf("%d %d",&m,&n);

if(m==n)
{
printf("enter elements of the matrix\n");

for(i=0;i<m;i++)
{
for(j=0;j<n;j++)
{
scanf("%d",&a[i][j]);

printf("\n");

}

}

}
```

```
printf("the input matrix is\n\n\n\n");
```

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

```
{
```

```
    for(j=0;j<n;j++)
```

```
    {
```

```
        printf("%d\t",a[i][j]);
```

```
    }
```

```
    printf("\n\n\n\n\n\n");
```

```
}
```

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

```
{
```

```
    for(j=0;j<n;j++)
```

```
    {
```

```
        b[i][j]=a[j][i];
```

```
    }
```

```
}
```

```

printf("transpose of a matrix is\n\n\n\n\n");
for(i=0;i<m;i++)
{
    for(j=0;j<n;j++)
    {
        printf("%d\t",b[i][j]);

    }
    printf("\n\n\n\n\n");
}
for(i=0;i<m;i++)
{
    for(j=0;j<n;j++)
    {
        if(a[i][j]!=b[i][j])
        {
            temp=0;

            //printf("matrix is symmetric");
        }
    }
}

```

```
        }  
    }  
    if(temp==0)  
  
        printf("matrix is not symmetric");  
    else  
        printf("matrix is symmetric");  
}  
else  
    printf("symmetric matrix shud be a square matrix");  
}
```

C - Strings

Strings are actually one-dimensional array of characters terminated by a **null** character '\0'. Thus a null-terminated string contains the characters that comprise the string followed by a **null**.

The following declaration and initialization create a string consisting of the word "Hello". To hold the null character at the end of the array, the size of the character array containing the string is one more than the number of characters in the word "Hello."

```
char greeting[6] = {'H', 'e', 'l', 'l', 'o', '\0'};
```

If you follow the rule of array initialization then you can write the above statement as follows –

```
char greeting[] = "Hello";
```

Following is the memory presentation of the above defined string in C/C++ –

Index	0	1	2	3	4	5
Variable	H	e	l	l	o	\0
Address	0x23451	0x23452	0x23453	0x23454	0x23455	0x23456

Actually, you do not place the *null* character at the end of a string constant. The C compiler automatically places the '\0' at the end of the string when it initializes the array. Let us try to print the above mentioned string –

```
#include <stdio.h>

int main () {

    char greeting[6] = {'H', 'e', 'l', 'l', 'o', '\0'};

    printf("Greeting message: %s\n", greeting );

    return 0;
```

```
}
```

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

```
Greeting message: Hello
```

C supports a wide range of functions that manipulate null-terminated strings –

Sr.No.	Function & Purpose
1	strcpy(s1, s2); Copies string s2 into string s1.
2	strcat(s1, s2); Concatenates string s2 onto the end of string s1.
3	strlen(s1); Returns the length of string s1.
4	strcmp(s1, s2); Returns 0 if s1 and s2 are the same; less than 0 if s1<s2; greater than 0 if s1>s2.
5	strchr(s1, ch); Returns a pointer to the first occurrence of character ch in string s1.
6	strstr(s1, s2); Returns a pointer to the first occurrence of string s2 in string s1.

String Handling Functions in C

C programming language provides a set of pre-defined functions called **string handling functions** to work with string values. The string handling functions are defined in a header file called **string.h**. Whenever we want to use any string handling function we must include the header file called **string.h**.

The following table provides most commonly used string handling function and their use...

Function	Syntax (or) Example	Description
strcpy()	strcpy(string1, string2)	Copies string2 value into string1
strncpy()	strncpy(string1, string2, 5)	Copies first 5 characters string2 into string1
strlen()	strlen(string1)	returns total number of characters in string1
strcat()	strcat(string1,string2)	Appends string2 to string1
strncat()	strncpy(string1, string2, 4)	Appends first 4 characters of string2 to string1
strcmp()	strcmp(string1, string2)	Returns 0 if string1 and string2 are the same; less than 0 if string1<string2; greater than 0 if string1>string2
strncmp()	strncmp(string1, string2, 4)	Compares first 4 characters of both string1 and string2
strcmpi()	strcmpi(string1,string2)	Compares two strings, string1 and string2 by ignoring case (upper or lower)

Function	Syntax (or) Example	Description
stricmp()	stricmp(string1, string2)	Compares two strings, string1 and string2 by ignoring case (similar to strcmpi())
strlwr()	strlwr(string1)	Converts all the characters of string1 to lower case.
strupr()	strupr(string1)	Converts all the characters of string1 to upper case.
strdup()	string1 = strdup(string2)	Duplicated value of string2 is assigned to string1
strchr()	strchr(string1, 'b')	Returns a pointer to the first occurrence of character 'b' in string1
strrchr()	'strrchr(string1, 'b')	Returns a pointer to the last occurrence of character 'b' in string1
strstr()	strstr(string1, string2)	Returns a pointer to the first occurrence of string2 in string1
strset()	strset(string1, 'B')	Sets all the characters of string1 to given character 'B'.
strnset()	strnset(string1, 'B', 5)	Sets first 5 characters of string1 to given character 'B'.
strrev()	strrev(string1)	It reverses the value of string1

The following example uses some of the above-mentioned functions –

The following example uses Strcpy(), strcat() and strlen()

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

    char str1[12] = "Hello";
    char str2[12] = "World";
    char str3[12];
    int len ;

    /* copy str1 into str3 */
    strcpy(str3, str1);
    printf("strcpy( str3, str1) : %s\n", str3 );

    /* concatenates str1 and str2 */
    strcat( str1, str2);
    printf("strcat( str1, str2):  %s\n", str1 );

    /* total length of str1 after concatenation */
    len = strlen(str1);
    printf("strlen(str1) : %d\n", len );

    return 0;
}
```

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

```
strcpy( str3, str1) : Hello
strcat( str1, str2):  HelloWorld
strlen(str1) : 10
```

Structures in C

In C programming language, a structure is a collection of elements of the different data type. The structure is used to create user-defined data type in the C programming language. As the structure used to create a user-defined data type, the structure is also said to be “user-defined data type in C”. In other words, a structure is a collection of non-homogeneous elements. Using structure we can define new data types called user-defined data types that holds multiple values of the different data type. The formal definition of structure is as follows...

Structure is a collection of different type of elements under a single name that acts as user defined data type in C.

Generally, structures are used to define a record in the c programming language. Structures allow us to combine elements of a different data type into a group. The elements that are defined in a structure are called members of structure.

How to create structure?

To create structure in c, we use the keyword called "**struct**". We use the following syntax to create structures in c programming language.

```
struct <structure_name>  
{  
    data_type member1;  
    data_type member2, member3;  
    .  
    .  
}
```

Following is the example of creating a structure called Student which is used to hold student record.

Creating Structure in C

```
struct student
{
char stud_name[30];
int roll_number;
float percentage;
}
```

Important Points to be Remembered

Every structure must be terminated with semicolon symbol (;).
"struct" is a keyword, it must be used in lowercase letters only.

Creating and Using structure variables

In a C programming language, there are two ways to create structure variables. We can create structure variable while defining the structure and we can also create after terminating structure using struct keyword.

To access members of a structure using structure variable, we use dot (.) operator. Consider the following example code...

Creating and Using structure variables in C

```
struct Student
{
    char stud_name[30];
    int roll_number;
    float percentage;
} stud_1 ;           // while defining structure

void main(){
    struct Student stud_2; // using struct keyword

    printf("Enter details of stud_1 : \n");
    printf("Name : ");
    scanf("%s", stud_1.stud_name);
```

```

printf("Roll Number : ");
scanf("%d", &stud_1.roll_number);
printf("Percentage : ");
scanf("%f", &stud_1.percentage);

printf("***** Student 1 Details *****\n");
printf("Name of the Student : %s\n", stud_1.stud_name);
printf("Roll Number of the Student : %i\n", stud_1.roll_number);
printf("Percentage of the Student : %f\n", stud_1.percentage);
}

```

In the above example program, the structure variable "**stud_1**" is created while defining the structure and the variable "**stud_2**" is created using struct keyword. Whenever we access the members of a structure we use the dot (.) operator.

Memory allocation of Structure

When the structures are used in the C programming language, the memory does not allocate on defining a structure. The memory is allocated when we create the variable of a particular structure. As long as the variable of a structure is created no memory is allocated. The size of memory allocated is equal to the sum of memory required by individual members of that structure. In the above example program, the variables **stud_1** and **stud_2** are allocated with 36 bytes of memory each.

```

struct Student{
    char stud_name[30]; ————— 30 bytes
    int roll_number; ————— 02 bytes
    float percentage; ————— 04 bytes
};
                                sum = 36 bytes

```

Here the variable of **Student** structure is allocated with 36 bytes of memory.

Important Points to be Remembered

All the members of a structure can be used simultaneously.

Until variable of a structure is created no memory is allocated. The

memory required by a structure variable is sum of the memory required by individual members of that structure.

Unions in C

In C programming language, the union is a collection of elements of the different data type. The union is used to create user-defined data type in the C programming language. As the union used to create a user-defined data type, the union is also said to be “user-defined data type in C”. In other words, the union is a collection of non-homogeneous elements. Using union we can define new data types called user-defined data types that holds multiple values of the different data type. The formal definition of a union is as follows...

Union is a collection of different type of elements under a single name that acts as user defined data type in C.

Generally, unions are used to define a record in the c programming language. Unions allow us to combine elements of a different data type into a group. The elements that are defined in a union are called members of union.

How to create union?

To create union in c, we use the keyword called "union". We use the following syntax to create unions in c programming language.

```
union <structure_name>  
{  
    data_type member1;  
    data_type member2, member3;  
    .  
    .  
};
```

Following is the example of creating a union called Student which is used to hold student record.

Creating union in C

```

union Student
{
    char stud_name[30];
    int roll_number;
    float percentage;
} ;

```

Important	Points	to	be	Remembered
Every	union	must	terminated	with
"union" is a keyword, it must be used in lowercase letters only.				

Creating and Using union variables

In a c programming language, there are two ways to create union variables. We can create union variable while the union is defined and we can also create after terminating union using keyword. TO access members of a union using union variable, we use dot (.) operator. Consider the following example code...

Creating and Using union variables in C

```

union Student
{
    char stud_name[30];
    int roll_number;
    float percentage;
} stud_1 ;           // while defining union

```

```

void main(){
    union Student stud_2; // using union keyword

    printf("Enter details of stud_1 : \n");
    printf("Name : ");
    scanf("%s", stud_1.stud_name);
    printf("Roll Number : ");
    scanf("%d", &stud_1.roll_number);
}

```

```

printf("Percentage : ");
scanf("%f", &stud_1.percentage);

printf("***** Student 1 Details *****\n");
printf("Name of the Student : %s\n", stud_1.stud_name);
printf("Roll Number of the Student : %i\n", stud_1.roll_number);
printf("Percentage of the Student : %f\n", stud_1.percentage);
}

```

In the above example program, the union variable "**stud_1**" is created while defining the union and the variable "**stud_2**" is created using union keyword. Whenever we access the members of a union we use the dot (.) operator.

Memory allocation of Union

When the unions are used in the C programming language, the memory does not allocate on defining union. The memory is allocated when we create the variable of a particular union. As long as the variable of a union is created no memory is allocated. The size of memory allocated is equal to the maximum memory required by an individual member among all members of that union. In the above example program, the variables **stud_1** and **stud_2** are allocated with 30 bytes of memory each.

```

union Student{
    char stud_name[30]; ————— 30 bytes
    int roll_number; ————— 02 bytes
    float percentage; ————— 04 bytes
};
                                max = 30 bytes

```

Here the variable of **Student** union is allocated with 30 bytes of memory and it is shared by all the members of that union.

POINTERS

INTRODUCTION

Pointers are one of the derived types in C. One of the powerful tool and easy to use once they are mastered.

Some of the advantages of pointers are listed below:

- ❖ A pointer enables us to access a variable that is defined outside the function. Pointers are more efficient in handling the data tables.
- ❖ Pointers reduce the length and complexity of a program.
- ❖ The use of a pointer array to character strings save data storage space in memory.

The real power of C lies in the proper use of pointers.

POINTER CONCEPTS

The basic data types in C are int, float, char double and void. Pointer is a special data type which is derived from these basic data types.

There are three concepts associated with the pointers are,

- ❖ Pointer Constants
- ❖ Pointer Values
- ❖ Pointer Variables

POINTER CONSTANT

- ❖ As we know, computers use their memory for storing the instructions of a program, as well as the values of the variables that are associated with it.
- ❖ The computer's memory is a sequential collection of 'storage cells'.
- ❖ Each cell can hold one byte of information, has a unique number associated with it called as 'address'.
- ❖ The computer addresses are numbered consecutively, starting from zero. The last address depends on the memory size.

- ❖ Let us assume the size of the memory is 64K then,

The total memory locations = 64K
= 64 * 1K
= 64 * 1024 bytes
= 65536 bytes (locations)

- ❖ So, here the last address is 65535(started with 0).
- ❖ Physically they are divided into even bank and odd bank.
- ❖ Even bank is set of memory locations with even addresses. Like 0, 2, 4, 6.....65534.
- ❖ Odd bank is set of memory locations with odd addresses. Like 1, 3, 565535.

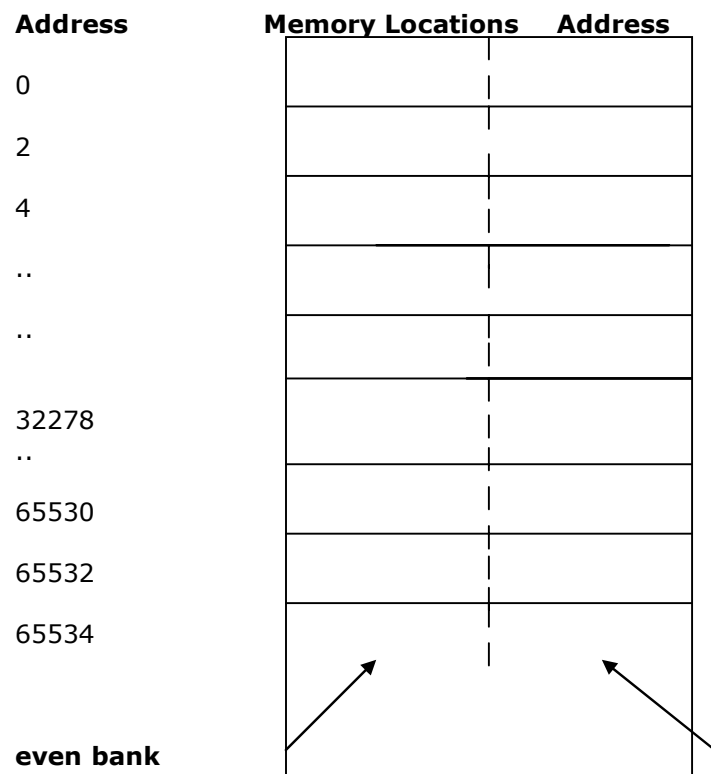


Figure: 3.1 Memory Organization.

- ❖ These memory addresses are called **pointer constants**.
- ❖ We cannot change them, but we can only use them to store data values.
- ❖ For example, in the above memory organization, the addresses ranging from 0 to 65535 are known as pointer constants.
- ❖ Remember one thing, the address of a memory location is a pointer constant and cannot be changed .

POINTER VALUE

Whenever we declare a variable, the system allocates, an appropriate location to hold the value of the variable somewhere in the memory.

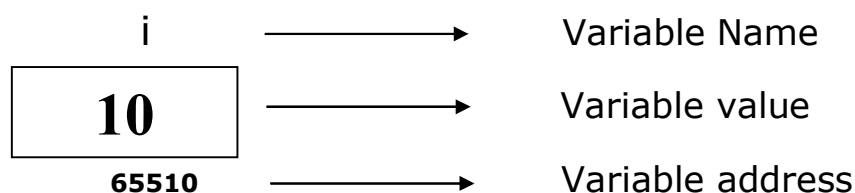
Consider the following declaration,

int i=10;

This declaration tells the C compiler to perform the following activities:

- ❖ Reserve **space** in the memory to hold the integer value.
- ❖ Associate the name **i** with this memory location.
- ❖ Store the value **10** at this location.

We can represent i's location in the memory by the following memory map:



Pointer Values

- ❖ Memory is divided into number of storage cells called locations.
- ❖ Out of these the addresses, the system assigns some addresses of the memory locations to the variables.
- ❖ These memory locations assigned to the variables by the system are called **pointer values**.
- ❖ For example, the address 65510 which is assigned to the variable i is a pointer value.

The & Operator

- ❖ The address of the variable cannot be accessed directly. The address can be obtained by using address operator(**&**) in C language.
- ❖ The address operator can be used with any variable that can be placed on the left side of an assignment operator.
- ❖ The format specifier of address is %u(unsigned integer),the reason is addresses are always positive values. We can also use %x to know the address of a variable.
- ❖ Example, to know the address of variable n, just use &n.

Note: Constants, expressions, and array name cannot be placed on the left side of the assignment and hence accessing address is invalid for constants, array names and expressions.

The following are illegal use of address Operator.

&125 (Pointing at constant)

int a[10];
&a (pointing to array name)

&(x+y) (pointing at expressions)

POINTER VARIABLE

- ❖ A variable which holds the address of some other variable is called pointer variable.
- ❖ A pointer variable should contain always the address only.

The * Operator

- ❖ It is called as '**Value at address**' operator. It returns the value stored at a particular address.
- ❖ It is also Known as **Indirection** or **Dereferencing Operator**

ACCESSING A VARIABLE THROUGH POINTER

For accessing the variables through pointers, the following sequence of operations have to be performed, to use pointers.

1. Declare an ordinary variable.
2. Declare a pointer variable.
3. Initialize a pointer variable (Provide link between pointer variable and ordinary variable).
4. Access the value of a variable using pointer variable.

We already familiar with the declaration and initialization of variable. Now we will discuss the remaining here.

Declaring a pointer variable

In C , every variable must be declared before they are used. Since the pointer variables contain address that belongs to a separate data type, they must be declared as pointers before we use them.

The syntax for declaring a pointer variable is as follows,

`data type *ptr_name;`

This tells the compiler three things about the variable ptr_name.

1. The asterisk(*) tells that the variable ptr_name is a pointer variable.
2. ptr_name needs a memory location.
3. ptr_name points to a variable of type data type.

For example,

int *pi;

declares the variable p as a pointer variable that points to an integer data type.

Remember that the type **int** refers to the data type of the variable being pointed by **pi**.

Initializing Pointers

- ❖ Once a pointer variable has been declared, it can be made to point to a variable using statement such as

`ptr_name=&var;`

- ❖ Which cause **ptr_name** to point to **var**. Now **ptr_name** contains the address of **var**. This is known as pointer initialization.
- ❖ Before a pointer is initialized it should not be used.

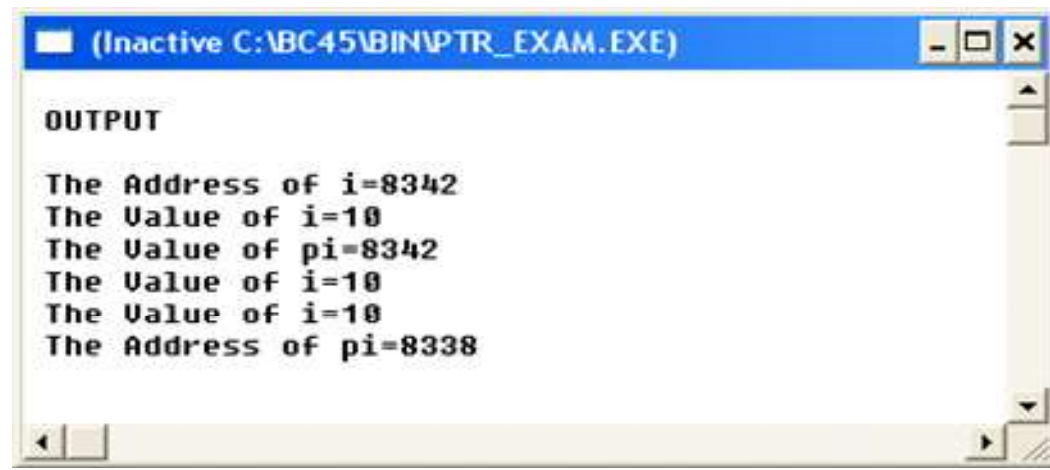
Access the value of a variable using pointer variable

Once a pointer variable has been assigned the address of a variable, we can access the value of a variable using the pointer. This is done by using the indirection operator(*).

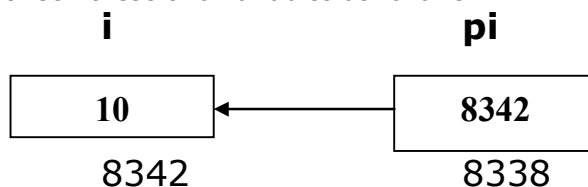
`*ptr_name`

Example1

```
// C Program illustrating accessing of variables using pointers
#include<stdio.h>
int main()
{
    int i=10;
    int *pi;
    pi=&i;
    printf("\n OUTPUT\n");
    printf("\n The Address of i=%u",&i);
    printf("\n The Value of i=%d",i);
    printf("\n The Value of pi=%u",pi);
    printf("\n The Value of i=%d",*(&i));
    printf("\n The Value of i=%d",*pi);
    printf("\n The Address of pi=%u",&pi);
    return 0;
}
```



The above program illustrates how to access the variable using pointers. After finding the first statement `i=10`, the compiler creates a variable `i` with a value of 10 at a memory location. Then coming to line 2 and 3 a pointer variable `pi` is created and initialized with the address of the `i` variable. Then the compiler automatically provides a link between these two variables as follows.



Note: Pointer variable always points to a address of the another variable .Following statements are not valid with respect to pointers.

```
int i=10, k, *pi=&i;
```

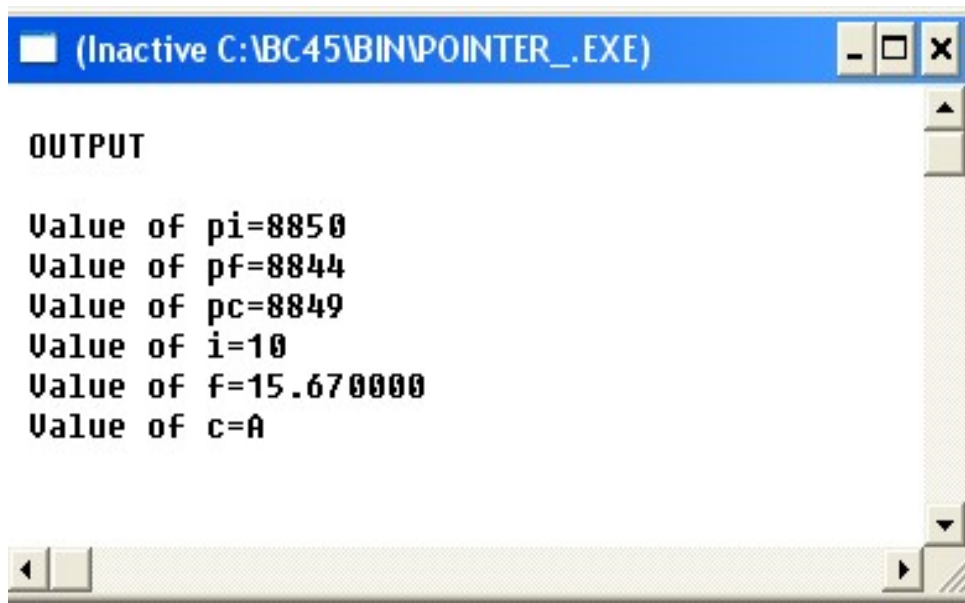
```
k=pi; // pointer value cannot be accessed by integer
```

```
pi=65506(constant); // we cannot directly assign a value to a pointer variable
```

Example2

The following code illustrates how to declare `int`, `char` and `float` pointers. Here we have declared three variables of type `int`, `float` and `char` ,also three pointer variables points to `int`, `float` and `char`. Remember here `pf` points to the value of type `float` but its type is unsigned integer only.

```
#include<stdio.h>
int main()
{
    int i=10;
    char c='A';
    float f=15.67;
    int *pi; /*Pointer to an integer */
    float *pf; /*pointer to a float number */
    char *pc; /*pointer to a character; */
    pi=&i;
    pf=&f; /*initialization of pointer variables */
    pc=&c;
    printf("\n OUTPUT \n");
    printf("\n Value of pi=%u",pi);
    printf("\n Value of pf=%u",pf);
    printf("\n Value of pc=%u",pc);
    printf("\n Value of i=%d",*pi);
    printf("\n Value of f=%f",*pf);
    printf("\n Value of c=%c",*pc);
    return 0;
}
```



Declaration versus Redirection:

- ❖ When an asterisk is used for declaration, it is associated with a type.
- ❖ Example: `int* pa;`
 `int* pb;`
- ❖ On the other hand, we also use the asterisk for redirection.
When used for redirection, the asterisk is an operator that redirects the operation
from the pointer variable to a data variable.
- ❖ Example: `Sum = *pa + *pb;`

Dangling Pointers

A pointer variable should contain a valid address. A pointer variable which does not contain a valid address is called dangling pointer.

For example, consider the following declaration,

```
int *pi;
```

This declaration indicates that pi is a pointer variable and the corresponding memory location should contain address of an integer variable. But, the declaration will not initialize the memory location and memory contains garbage value as shown in below.

Note: We cannot use a pointer variable to the register variable. The reason is that, user does not know the address of the register variable. So we are not able to use pointer variable on register variables.

Example To Demonstrate Working of Pointers

```
/* Source code to demonstrate, handling of pointers in C program */
#include <stdio.h>
int main()
{
    int *pc,c;
    c=22;
    printf("Address of c:%d\n",&c);
    printf("Value of c:%d\n\n",c);
    pc=&c;
    printf("Address of pointer pc:%d\n",pc);
    printf("Content of pointer pc:%d\n\n",*pc);
    c=11;
    printf("Address of pointer pc:%d\n",pc);
    printf("Content of pointer pc:%d\n\n",*pc);
    *pc=2;
    printf("Address of c:%d\n",&c);
    printf("Value of c:%d\n\n",c);
    return 0;
}
```

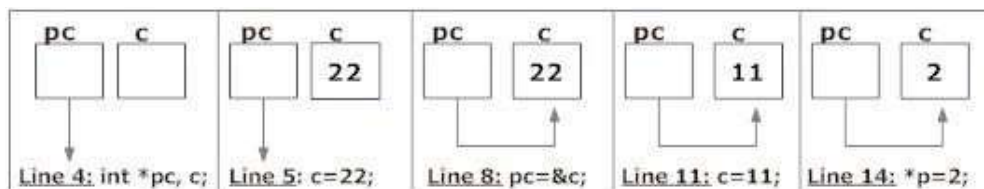
Output

Address of c: 2686784
Value of c: 22

Address of pointer pc: 2686784
Content of pointer pc: 22

Address of pointer pc: 2686784
Content of pointer pc: 11

Address of c: 2686784
Value of c: 2



Explanation of program and figure

1. Code `int *pc, p;` creates a pointer `pc` and a variable `c`. Pointer `pc` points to some address and that address has garbage value. Similarly, variable `c` also has garbage value at this point.
2. Code `c=22;` makes the value of `c` equal to 22, i.e., 22 is stored in the memory location of variable `c`.
3. Code `pc=&c;` makes pointer, point to address of `c`. Note that, `&c` is the address of variable `c` (because `c` is normal variable) and `pc` is the address of `pc` (because `pc` is the pointer variable). Since the address of `pc` and address of `c` is same, `*pc` (value of pointer `pc`) will be equal to the value of `c`.
4. Code `c=11;` makes the value of `c`, 11. Since, pointer `pc` is pointing to address of `c`. Value of `*pc` will also be 11.
5. Code `*pc=2;` change the address pointed by pointer `pc` to change to 2. Since, address of pointer `pc` is same as address of `c`, value of `c` also changes to 2.

Review Questions:

1. Write a C program to read in an array of integers. Instead of using subscripting, however, employ an integer pointer that points to the elements currently being read in and which is incremented each time.
2. Write a C program using pointers to read in an array of integers and print its elements in reverse order.
3. Write a C program to arrange the given numbers in ascending order using pointers.
4. Write a 'C' program to find factorial of a given number using pointers.
5. a) What is a pointer variable? How is a pointer variable different from an ordinary variable.
b) Distinguish between address operator and dereferencing operator.
c) Write a C Program to illustrate the use of indirection operator "*" to access the value pointed by a pointer.
6. Explain the uses of Pointers with an example.

Snippet Programs:

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

int main()
{
    int i=3, *j, k;
    j = &i;
    printf("%d\n", i**j*i+*j);
    return 0;
}
```

```

2.  #include<stdio.h>
    int main()
    {
        int x=30, *y, *z;
        y=&x; /* Assume address of x is 500 and integer is 4 byte */
        z=y;
        *y++=*z++;
        x++;
        printf("x=%d, y=%d, z=%d\n", x, y, z);
        return 0;
    }

```

```

3.  #include<stdio.h>
    int main()
    {
        char *str;
        str = "%s";
        printf(str, "K\n");
        return 0;
    }

```

POINTERS AND FUNCTIONS

- ❖ Pointers can be used to pass addresses of variables to called functions, thus allowing the called function to alter the values stored there.
- ❖ We looked earlier at a swap function that did not change the values stored in the main program because only the values were passed to the function swap.
- ❖ This is known as "call by value".
- ❖ Here we are going to discuss how to pass the address.

Call by Reference

Instead of passing the values of the variables to the called function, we pass their addresses, so that the called function can change the values stored in the calling routine. This is known as "call by reference", since we are referencing the variables.

Here the addresses of actual arguments in the calling function are copied into formal arguments of the called function. Here The formal parameters should be declared as pointer variables to store the address.

The following shows the swap function modified from a "call by value" to a "call by reference". Note that the values are now swapped when the control is returned to main function.

```

#include <stdio.h>
void swap ( int *pa, int *pb );
int main ( )
{
    int a = 5, b = 6;
    printf("a=%d b=%d\n",a,b) ;
    swap (&a, &b) ;
    printf("a=%d b=%d\n",a,b) ;
    return 0 ;
}

void swap( int *pa, int *pb )
{
    int temp;
    temp= *pa; *pa= *pb; *pb = temp ;
    printf ("a=%d b=%d\n", *pa, *pb);
}

```

Results:

```

a=5 b=6
a=6 b=5
a=6 b=5

```

Observe the following points when the program is executed,

- ❖ The address of actual parameters a and b are copied into formal parameters pa and pb.
- ❖ In the function header of swap (), the variables a and b are declared as pointer variables.
- ❖ The values of a and b accessed and changed using pointer variables pa and pb.

Call by Value	Call by Reference
<i>When Function is called the values of variables are passed.</i>	When a function is called address of variables is passed.
<i>Formal parameters contain the value of actual parameters.</i>	Formal parameters contain the address of actual parameters.
<i>Change of formal parameters in the function will not affect the actual parameters in the calling function</i>	The actual parameters are changed since the formal parameters indirectly manipulate the actual parameters
<i>Execution is slower since all the values have to be copied into formal parameters.</i>	Execution is faster since only addresses are copied.

Table: 3.1 Difference between Call by Value and Call by Reference

FUNCTION RETURNING POINTERS

- ❖ The way function return an int, float and char, it can return a pointer.
- ❖ To make a function return a pointer it has to be explicitly mentioned in the calling function as well as in the function declaration.
- ❖ Three things should be done to avail the feature of functions return pointer.

1. Declaration of function returning pointer
2. Declaring pointer and assigning function call
3. Defining function returning pointer

- ❖ Syntax for declaration of function returning pointer

```
return_type *function_name (arguments);
```

This declaration helps the compiler to recognize that this function returns address.

- ❖ Now declare pointer variable and place the function call

```
ptr = function_name (arguments);
```

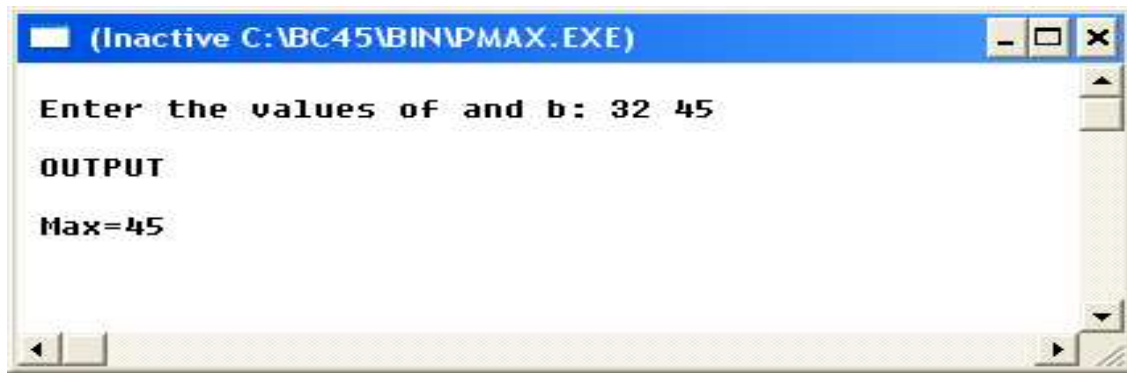
- ❖ After executing above statement ptr consisting of the address that is returned by the function. Remember the return type of the function and pointer type should match here.
- ❖ The function Definition returning pointer takes of the form,

```
return_type *function_name (arguments)
{
    // local declarations
    // executable statements

    return (&variable); Here don't forget to send address
                        with return statement.
}
```

Example:

```
#include<stdio.h>
int *max(int* ,int* );
int main()
{
    int a,b;
    int *ptr;
    printf("\n Enter the values of a and b: ");
    scanf("%d%d",&a,&b);
    ptr=max(&a,&b);
    printf("\n OUTPUT \n");
    printf("\n Max=%d",*ptr);
    getch();
}
int *max(int*pa,int*pb)
{
    if(*pa>*pb)
        return pa;
    else
        return pb;
}
```



The execution of the program as follows,

- ❖ Execution of the program starts at main.
- ❖ Two variables a and b are created and initialized at run-time.
- ❖ A pointer variable is created and initialized with the return value of the function max ().
- ❖ Once the control is transferred from function main () to max (), it got executed and returns the pointer value to main().
- ❖ Here we are having the address of the maximum variable address to display it just use indirection operator (*).

Note: function return pointer does not have any advantage except in the handling of strings.

POINTERS TO FUNCTIONS

Pointer to a function (also known as function pointer) is a very powerful feature of C. Function pointer provides efficient and elegant programming technique. Function pointers are less error prone than normal pointers since we will never allocate or de-allocate memory for the functions.

Every variable with the exception of register has an address. We have seen how we can refer variables of type char, int and float. Through their addresses, by using pointers.

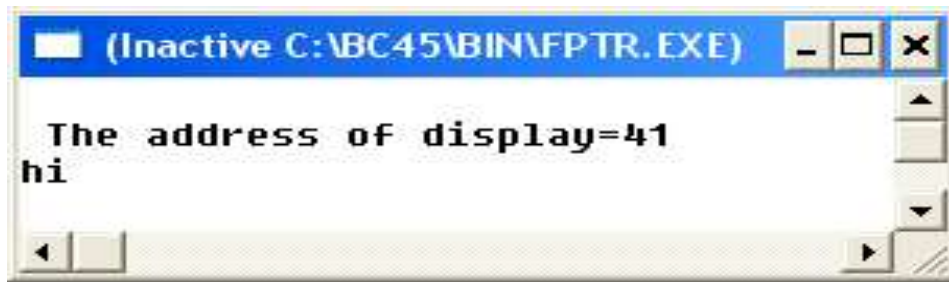
Functions exist in memory just like variables. C will allow you to define pointers to functions. Just like variables, a function name gives the starting address of function stored in memory.

The below code illustrate how to get the address of a function.

```
#include<stdio.h>

int main()
{
    void display();
    printf("\n The address of display=%u",display);
    getch();
}

void display()
{
    printf("hi");
}
```



DEFINING POINTERS TO FUNCTIONS

Like declaring pointer variables, we can define pointers to function variables and store the address. The below figure illustrate how function pointer can be represented.

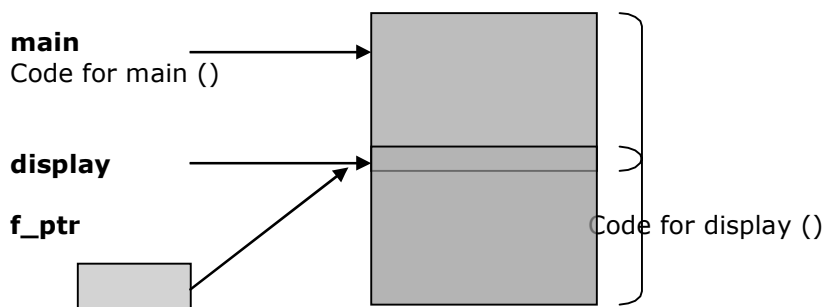


Figure: 3.3. Functions in Memory.

The syntax for declaring pointer to function as follows,

```
return_type (*f_ptr) (arguments);
```

Everything is same as function declaration except the braces for the name, to tell the compiler that this is a function pointer braces are required here and as usual for pointer declarations * is used.

Note that the return type of function pointer, number of arguments and type of arguments must match with the normal function.

The next after the declaration is calling the function using function pointer. before calling takes place we must initialize the function pointer with the address of the function.

The syntax for this assignment,

```
f_ptr=function_name;
```

After this assignment we need to call the function, the syntax associated with the function call is as follows,

```
(*f_ptr)(argument's);
```

This is another way of calling the function. There are no changes in the declaration of the function body.

The below program simulates a simple calculator using function pointers.

```
#include<stdio.h>

int add(int* ,int* );
int sub(int* ,int* );
int mul(int* ,int* );
float div(int* ,int* );
int (*fadd)(int* ,int* );
int (*fsub)(int* ,int* );
int (*fmul)(int* ,int* );
float (*fdiv)(int* ,int* );

int main()
{
    int x,y,ch;
    char c;
    fadd=add;
    fsub=sub;
    fmul=mul;
    fdiv=div;
    printf("\nEnter the values of two operands :");
    scanf("%d%d",&x,&y);
    printf("\n");
    do
    {
        printf("          1.ADDITION          \n");
        printf("          2.SUBTRACTION         \n");
        printf("          3.MULTIPLICATION      \n");
        printf("          4.DIVISION            \n");
        printf("\nEnter your choice :");
        scanf("%d",&ch);
        switch(ch)
        {
            case 1:
```

```

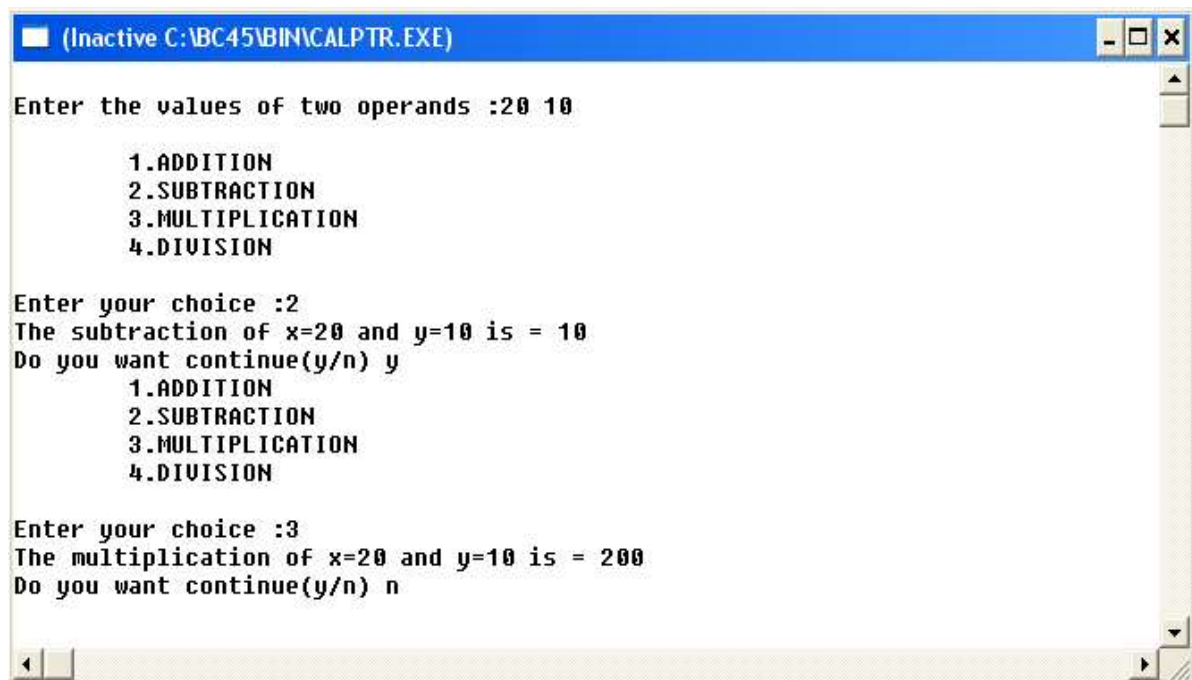
    printf("The addition of x=%d and y=%d is = %d",x,y,(*fadd)(&x,&y));
    break;
case 2:
    printf("The subtraction of x=%d and y=%d is = %d",x,y,(*fsub)(&x,&y));
    break;
case 3:
    printf("The multiplication of x=%d and y=%d is = %d",x,y,(*fmul)(&x,&y));
    break;
case 4:
    printf("The division of x=%d and y=%d is = %f",x,y,(*fdiv)(&x,&y));
    break;
}
printf("\nDo you want continue(y/n) ");
fflush(stdin);
scanf("%c",&c);
    } while (c=='y' || c=='Y');
    getch();
}

int add(int *a,int *b)
{
    return(*a+*b);
}

int sub(int *a,int *b)
{
    return(*a-*b);
}
int mul(int *a,int *b)
{
    return(*a * (*b));
}

float div(int *a,int *b)
{
    return(*a/(*b));
}

```



Review Questions:

1. Write a 'C' program to find factorial of a given number using pointers.
2. a) How to use pointers as arguments in a function? Explain through an example.
b) Write a 'C' function using pointers to exchange the values stored in two locations in the memory.
3. Explain pointer to function and function returning pointer with example.

Snippet Programs:

```
1. #include<stdio.h>
   void fun(void *p);
   int i;

   int main()
   {
       void *vptr;
       vptr = &i;
       fun(vptr);
       return 0;
   }
   void fun(void *p)
   {
       int **q;
       q = (int**) &p;
       printf("%d\n", **q);
   }
```

```

2. #include<stdio.h>
   int *check(static int, static int);

   int main()
   {
       int *c;
       c = check(10, 20);
       printf("%d\n", c);
       return 0;
   }
   int *check(static int i, static int j)
   {
       int *p, *q;
       p = &i;
       q = &j;
       if(i >= 45)
           return (p);
       else
           return (q);
   }

```

POINTER ARITHMETIC

The following operations can be performed on a pointer:

- ❖ Addition of a number to a pointer. Pointer can be incremented to point to the next locations.

Example:

```

int i=4 ,pi=&i; //(assume address of i=1000)
float j,*pj=&j;// (assume address of j=2000)
pi = pi + 1; // here pi incremented by (1*data type times)
pi = pi + 9; // pi = 1000 + (9*2) → 1018 address
pj = pj + 3; // pj=1018+(3*4)→1030 address

```

- ❖ Subtraction of a number from a pointer. Pointer can be decremented to point to the earlier locations.

Example:

```

int i=4,*pi=&i; //(assume address of i =1000)
char c, *pc=&c; // assume address of c = 2000
double d, *pd=&d; // assume address of d=3000
pi = pi-2; /* pi=1000-(2*2)=996 address */
pc = pc-5; /* pc=2000-(5*1)=1985 address
pd = pd-6; /* pd=3000-(6*8)=2952 address */

```

- ❖ Pointer variables may be subtracted from one another. This is helpful while finding array boundaries. Be careful while performing subtraction of two pointers.
- ❖ Pointer variables can be used in comparisons, but usually only in a comparison to **NULL**.
- ❖ We can also use increment/decrement operators with pointers this is performed same as adding/subtraction of integer to/from pointer.

The following operations cannot be performed on pointers.

- ❖ Addition of two pointers.
- ❖ Multiplication of a pointer with a constant, two pointers.
- ❖ Division of a pointer with a constant, two pointers.

POINTER EXPRESSIONS

Like other variables, pointer variables can be used in expressions. For example, if p1 and p2 are two valid pointers, then the following statements are valid.

```
a = *p1 + *p2;
sum = sum + *p1;
z = 10 / *p2;
f = *p1 * i;
```

Note: be careful while writing pointer expressions. The expression ***p++** will result in the increment of the address of p by data type times and points to the new value. Whereas the expression **(*p)++** will increment the value at the address. If you are not properly coded you will get some unwanted result.

NULL Pointer

- ❖ If wish to have a pointer that points to "nowhere", should make this explicit by assigning it to NULL.
- ❖ If it is too early in the code to assign a value to a pointer, then it is better to assign NULL (i.e., \0 or 0).

```
double *pval1 = NULL;
double *pval2 = 0;
```

- ❖ The integer constants 0 and 0L are valid alternatives to NULL, but the symbolic constant is (arguably) more readable.
- ❖ A NULL pointer is defined as a special pointer.
- ❖ It can be used along with memory management functions.

POINTERS TO POINTERS

- ❖ It is possible to make a pointer to point to another pointer variable. But the pointer must be of a type that allows it to point to a pointer.
- ❖ A variable which contains the address of a pointer variable is known as **pointer to pointer**.
- ❖ Its major application is in referring the elements of the two dimensional array.
- ❖ Syntax for declaring pointer to pointer,

```
data type **ptr_ptr;
```

- ❖ This declaration tells compiler to allocate a memory for the variable **ptr_ptr** in which address of a **pointer variable** which points to value of type data type can be stored.
- ❖ Syntax for initialization

```
ptr_ptr=&ptr_name;
```

- ❖ This initialization tells the compiler that now ptr_ptr points to the address of a pointer variable.
- ❖ Accessing the element value,

```
**ptr_ptr;
```

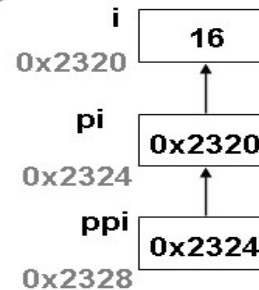
- ❖ It is equivalent to `*(&ptr_name);`

Example

```
#include <stdio.h>
int main(void)
{
    int i = 16;
    int *pi = &i;
    int **ppi;

    ppi = &pi;
    printf("%i", **ppi);
    return 0;
}
```

pp is a "pointer to" a
"pointer to an int"



The above program illustrates the use of pointers to pointers. Here, using two indirection operators the data item 16 can be accessed (i.e., ***pi** refers to **pi** and ****ppi** refers to **i**).

POINTER COMPATIBILITY

- ❖ We should not store the address of a data variable of one type into a pointer variable of another type.
- ❖ During assigning we should see that the type of data variable and type of the pointer variable should be same or compatible.
- ❖ Other wise it will result in unwanted output.
- ❖ The following program segment is wrong,

```
int i=10;

float *pf;

pf=&i; // data variable is integer and pointer
       variable is float
```

It is possible to use incompatible pointer types while assigning with type casting pointer.

Casting pointers

When assigning a memory address of a variable of one type to a pointer that points to another type it is best to use the cast operator to indicate the cast is intentional (this will remove the warning).

Example:

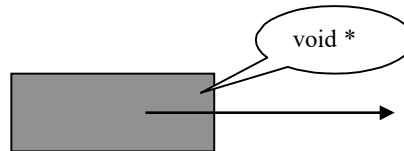
```
int V = 101;
float *P = (float *) &V; /* Casts int address to float * */
```

Removes warning, but is still a somewhat unsafe thing to do.

Void Pointer

- ❖ A pointer to void is a generic type that is not associated with a reference type.
- ❖ It is neither the address of a character nor an integer, nor a float nor any other type.
- ❖ It is compatible for assignment purposes only with all other pointer types.
- ❖ A pointer of any reference type can be assigned to a pointer to void type.
- ❖ A pointer to void type can be assigned to a pointer of any reference type.
- ❖ Certain library functions return void * results.
- ❖ No cast is needed to assign an address to a void * or from a void * to another pointer type.
- ❖ Where as a pointer to void can not be dereferenced unless it is cast.

void
Figure 3.2 pointer to void



❖ Example:

```
int V = 101;
float f=98.45;
void *G = &V;    /* No warning */
printf ("%d",*((int*)G)); /* Now it will display 101
float *P = G;    /* No warning, still not safe */
printf ("%f",*((float*)G)); /* Now it will display 98.45
```

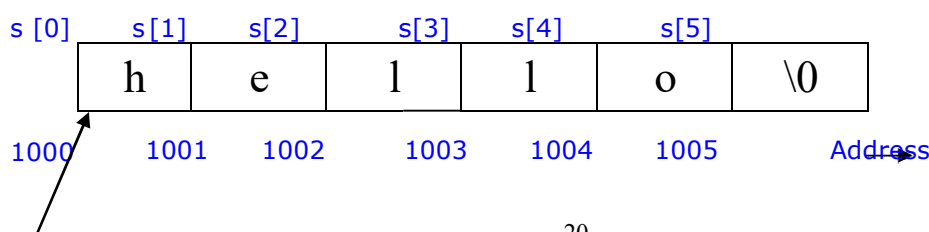
CHARACTER POINTER

Suppose we wish to store "Hello". We may either store it in a string or we may ask the C compiler to store it at some location in memory and assign the address of the string in a char pointer.

Consider the following declaration with string initialization,

```
char *p="hello";
```

Here the string length is 5 bytes. So the compiler allocates 6 bytes memory locations. Probably the characters are stored in the constant memory area of the computer, and the pointer p points to the base address as shown in the below figure 3.13.



p

1000

Figure 3.13: Initializing using Character Pointer

- ❖ We cannot assign a string to another. But, we can assign a char pointer to another char pointer.

```
Example: char *p1="hello";
char *p2;
p1=p2; //valid
printf ("%s", p1); //will print hello
```

- ❖ *p will refer to a particular character only, p will refer whole string.

POINTERS AND STRINGS

A string is a character array. so the name of the string itself is a pointer. Like referring array elements also string characters also referred with its name.

Example:

```
char s [] ="hello";
```

The following notations are used to refer individual characters

s[i] --> *(s+i) --> *(i+ s) all will point to the ith character in the given string.

We can also use pointer variable to refer the string elements.

```
char s [] ="hello";
char *ptr;
ptr=s; // now ptr points to the base address of the string.
```

then the following notations are same,

***ptr --> *(ptr+i) --> *(i+ptr)** will point value at the ith character.

Example: // illustrates displaying characters using pointer

```
#include<stdio.h>
void main ()
{
char s [] ="hello";
char *ptr;
ptr=s;
while (*ptr! ='\0')
{
printf (" %c",*ptr);
ptr++;
}
```

OUTPUT

h e l l o

Review Questions:

1. Write short notes on pointer to void.
2. Write short notes on Address Arithmetic.
3. Explain pointer to pointer with an example.

Snippet Programs:

1.

```
#include<stdio.h>
int main()
{
    static char *s[] = {"black", "white", "pink", "violet"};
    char **ptr[] = {s+3, s+2, s+1, s}, ***p;
    p = ptr;
    ++p;
    printf("%s", **p+1);
    return 0;
}
```
2.

```
#include<stdio.h>
int main()
{
    char str[20] = "Hello";
    char *const p=str;
    *p='M';
    printf("%s\n", str);
    return 0;
}
```
3.

```
#include<stdio.h>
int main()
{
    int ***r, **q, *p, i=8;
    p = &i;
    q = &p;
    r = &q;
    printf("%d, %d, %d\n", *p, **q, ***r);
    return 0;
}
```

POINTERS AND ARRAYS

- ❖ An array is a collection of similar elements stored in contiguous memory locations.
- ❖ When an array is declared, the compiler allocates a base address and sufficient amount of memory depending on the size and data type of the array.
- ❖ The base address is the location of the first element of the array.
- ❖ The compiler defines the array name as a constant pointer to the first element.

POINTERS AND ONE DIMENSIONAL ARRAY

Let us take the following declaration,

```
int num [5] = {1, 2, 3, 4, 5};
```

- ❖ After having this declaration, the compiler creates an array with name **num**, the elements are stored in contiguous memory locations, and each element occupies two bytes, since it is an integer array.
- ❖ The name of the array **num** gets the base address. Thus by writing ***num** we would be able to refer to the zeroth element of the array, that is 1.
- ❖ Then ***num** and ***(num+0)** both refer to the element 1. and ***(num+2)** will refer 3.
- ❖ When we have **num[i]**, the compiler internally converts it to ***(num+i)**.
- ❖ In this light the following notations are same.

$$\text{num}[i] \rightarrow *(num+i) \rightarrow *(i+num) \rightarrow i [\text{num}]$$

- ❖ Then we can also define a pointer and initialize it to the address of the first element of the array (base address).
- ❖ Example, for the above array we can have the following statement,

```
int *ptr=a; (or) int *ptr=&a[0];
```

- ❖ To refer the array elements by using pointer the following notations are used.

$$*ptr \rightarrow *(ptr+i) \rightarrow *(i+ptr) \rightarrow i [ptr]$$

- ❖ **ptr++** will point to the next location in the array.
- ❖ Accessing array elements by using pointers is always faster than accessing them by subscripts.
- ❖ The below figure shows the array element storage in the memory.

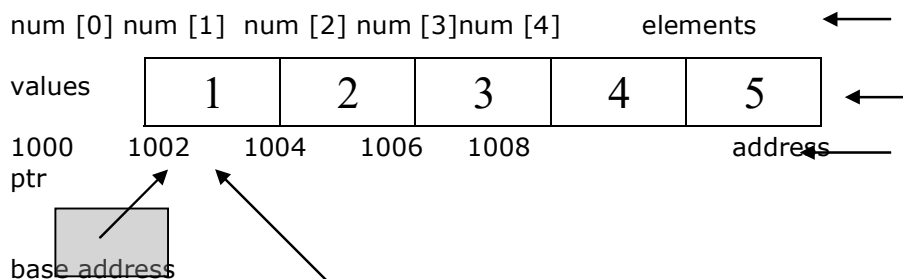


Figure 3.4 Storage representation of array

Example

```
#include<stdio.h>
int main()
{
    int num[5]={1,2,3,4,5};
    int i,*ptr;
    ptr=num;
    for(i=0;i<5;i++)
    {
        printf("\na[%d]=%d %d",i,*ptr,* (num+i));
        ptr++;
    }
    return 0;
}
```

```
a[0]=1 1
a[1]=2 2
a[2]=3 3
a[3]=4 4
a[4]=5 5
```

The above program illustrates displaying the array elements using pointers.

Note: Note that the array name num is a constant pointer points to the base address, then the increment of its value is illegal, num++ is invalid.

POINTERS AND TWO DIMENSIONAL ARRAYS

A two dimensional array is an array of one dimensional arrays. The important thing to notice about two-dimensional array is that, just as in a one-dimensional array, the name of the array is a pointer constant the first element of the array, however in 2-D array, the first element is another array.

Let us consider we have a two-dimensional array of integers. When we dereference the array name, we don't get one integer, we get an array on integers. In other words the dereference of the array name of a two-dimensional array is a pointer to a one-dimensional array. Here we require two indirections to refer the elements

Let us take the declaration

int a [3][4];

Then following notations are used to refer the two-dimensional array elements,

```
a----- > points to the first row
a+i ----- > points to ith row
*(a+i) ----- > points to first element in the ith row
*(a+i) +j ----- > points to jth element in the ith row
*(*(a+i)+j)----- >value stored in the ith row and jth column
```

	0	1	2	3
0	1	2	3	4
1	5	6	7	8
2	9	10	11	12

Example

```
#include<stdio.h>
int main()
{
    int a[3][4]={
        {1,2,3,4},
        {5,6,7,8},
        {9,10,11,12}
    };

    int i,j;
    for(i=0;i<3;i++)
    {
        for(j=0;j<4;j++)
        {
            printf("\nAddress of a[%d][%d]=%u value=%d",i,j,*(a+i)+j,*(*(a+i)+j));
        }
        printf("\n");
        printf("\n a=%u",a);
        printf("\n a+2=%u",a+2);
        printf("\n *(a+2)=%u",*(a+2));
        printf("\n *a[2]=%d",*a[2]);
        printf("\n *(a+2)+2=%u",*(a+2)+2);
        printf("\n *(*a+2)+2=%d",*(*(a+2)+2));
        getch();
    }
}
```

```
(Inactive C:\BC45\BINTWODARA.EXE)

Address of a[0][0]=8326 value=9327
Address of a[0][1]=8328 value=9327
Address of a[0][2]=8330 value=9327
Address of a[0][3]=8332 value=9327
Address of a[1][0]=8334 value=9327
Address of a[1][1]=8336 value=9327
Address of a[1][2]=8338 value=9327
Address of a[1][3]=8340 value=9327
Address of a[2][0]=8342 value=9327
Address of a[2][1]=8344 value=9327
Address of a[2][2]=8346 value=9327
Address of a[2][3]=8348 value=9327

a=8326
a+2=8342
*(a+2)=8342
**a+2=3
*a[2]=9
*(a+2)+2=8346
*(*(a+2)+2)=11
```

POINTERS AND THREE DIMENSIONAL ARRAYS

Three-dimensional array can be thought of array of two-dimensional array. To refer the elements here we require tree indirections.

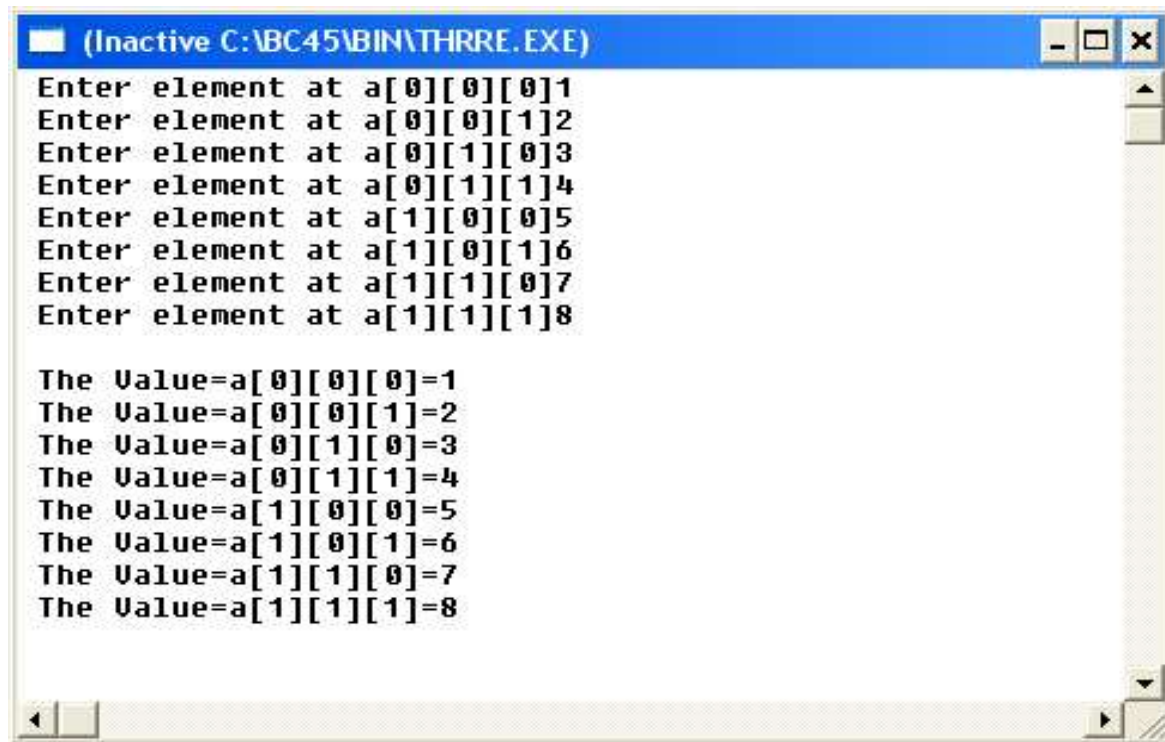
The notations are,

$*(*(a+i) + j) + k$ --> points to the address of kth dimension in ith row jth column

$*(*(*(a+i) + j) + k)$ --> value stored at kth dimension ith row jth column

Example

```
#include<stdio.h>
int main()
{
    int a[2][2][2],i,j,k;
    for(i=0;i<2;i++)
    {
        for(j=0;j<2;j++)
        {
            for(k=0;k<2;k++)
            {
                printf(" Enter element at a[%d][%d][%d]",i,j,k);
                scanf("%d",&*(*(a+i)+j)+k);
            }
        }
    }
    for(i=0;i<2;i++)
    {
        for(j=0;j<2;j++)
        {
            for(k=0;k<2;k++)
            {
                printf("\n The Value=a[%d][%d][%d]=%d",i,j,k,*(*(a+i)+j)+k);
            }
        }
    }
    getch();
}
```



```
(Inactive C:\BC45\BIN\THRE.EXE)
Enter element at a[0][0][0]1
Enter element at a[0][0][1]2
Enter element at a[0][1][0]3
Enter element at a[0][1][1]4
Enter element at a[1][0][0]5
Enter element at a[1][0][1]6
Enter element at a[1][1][0]7
Enter element at a[1][1][1]8

The Value=a[0][0][0]=1
The Value=a[0][0][1]=2
The Value=a[0][1][0]=3
The Value=a[0][1][1]=4
The Value=a[1][0][0]=5
The Value=a[1][0][1]=6
The Value=a[1][1][0]=7
The Value=a[1][1][1]=8
```

FUNCTIONS AND ARRAYS

To process arrays in large program, we have to be able to pass them to function. We can pass arrays in two ways,

- ❖ Passing individual elements
- ❖ Passing entire array elements

Passing individual elements

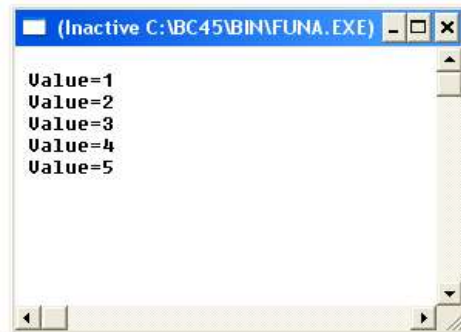
We can pass individual elements by either their data values or by passing their addresses.

1. Passing Data Values

- ❖ This is same as passing data values. Here we are passing an individual array element at a time to the function.
- ❖ The called function cannot tell whether the value it receives comes from an array, or a variable.

Example

```
#include<stdio.h>
void fun(int );
int main()
{
    int a[5]={1,2,3,4,5};
    int i;
    for(i=0;i<5;i++)
    {
        fun(a[i]);
    }
    return 0;
}
void fun(int x)
{
    printf("\n Value=%d",x);
}
```

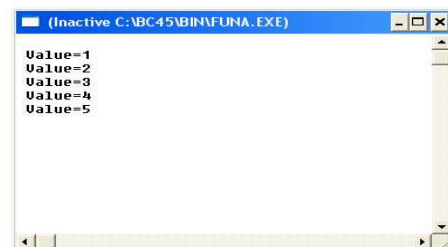


2. Passing Addresses of array elements

Here we are passing the individual elements address. the called function requires declaration of arguments as pointer variables.

Example

```
#include<stdio.h>
void fun(int* );
int main()
{
    int a[5]={1,2,3,4,5};
    int i;
    for(i=0;i<5;i++)
    {
        fun(&a[i]);
    }
    return 0;
}
void fun(int *px)
{
    printf("\n Value=%d",*px);
}
```

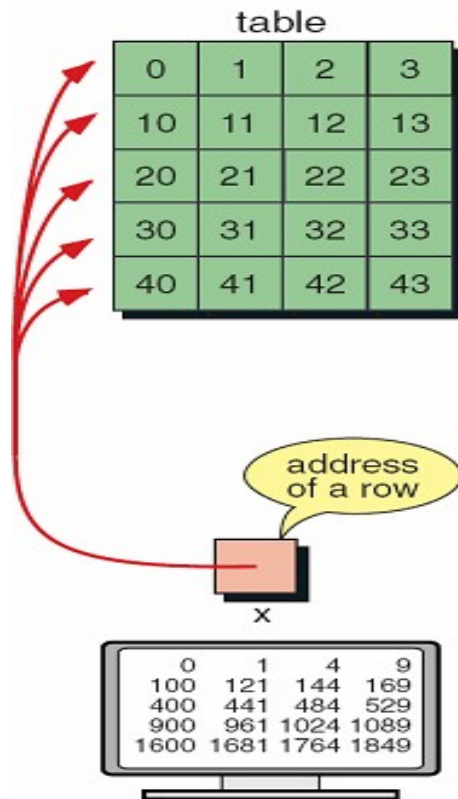


Passing the Entire Array

Here we see the first situation in which C does not pass values to a function. The reason for this change is that a lot of memory and time would be used in passing large arrays every time we wanted to use one in function.

For example, if an array containing 1200 elements were passed by value to a function, another 1200 elements would have to be allocated in the called function and each element would be copied from one array to the other. So, instead of passing the whole array, C passes the address of the array.

In C, the name of the array value is address of the first element in the array. When we pass the name, it allows the called function to refer to the array back in the calling function. Passing both two dimensional array and one dimensional are same but subscripts are changed.



```
#include<stdio.h>
void square (int [] [4]);
int main ()
{
    int a[5][4]={
        {0, 1, 2, 3},
        {10, 11, 12, 13},
        {20, 21, 22, 23},
        {30, 31, 32, 33},
        {40, 41, 42, 43}
    };

    square(a);
    return 0;
}
void square(int x[5][4])
{
    int i,j;
    for(i=0;i<5;i++)
    {
        for(j=0;j<4;j++)
        {
            printf ("\t%d",x[i][j]*x[i][j]);
        }
        printf("\n");
    }
}
```

The above program find the squares of the elements of the array .Here we passed the name of the array as an argument to the function an in the called function the formal argument is created and points to the elements of the calling function argument array.

POINTERS AND STRINGS

A string is a character array. so the name of the string it self is a pointer. Like referring array elements also string characters also refereed with its name.

Example:

```
char s [] ="hello";
```

The following notations are used to refer individual characters
s[i] --> *(s+i) --> *(i+ s) all will point to the ith character in the given string.
 We can also use pointer variable to refer the string elements.

```
char s [ ]="hello";
char *ptr;
ptr=s; // now ptr points to the base address of the string.
```

then the following notations are same,

***ptr --> *(ptr+i) --> *(i+ptr)** will point value at the ith character.

ARRAY OF POINTERS

- ❖ A pointer variable always contains an address; an array of pointers would be nothing but a collection of addresses.
- ❖ The addresses present in the array of pointers can be addresses of variables or addresses of array elements or any other addresses.
- ❖ The major application of this is in handling data tables efficiently and table of strings.
- ❖ All rules that apply to an ordinary array apply to the array of pointers as well.
- ❖ The Syntax for declaration of array of pointers as follows,

data type *arr_ptr [size];

- ❖ This declaration tells the compiler arr_ptr is an array of addresses, pointing to the values of data type.
- ❖ Then initialization can be done same as array element initialization. Example **arr_ptr [3] =&var**, will initialize 3rd element of the array with the address of var.
- ❖ The dereferencing operator is used as follows

***(arr_ptr [index])** --> will give the value at particular address.

- ❖ Look at the following code **array of pointers to ordinary Variables**

Example	OUTPUT
<pre>#include<stdio.h> void main () { int i=1, j=2, k=3, l=4, x; int *arr [4]; //declaration of array of pointers arr [0] =&i; //initializing 0th element with address of i arr [1] =&j; //initializing 1st element with address of j arr [2] =&k; //initializing 2nd element with address of k arr [3] =&l; //initializing 3rd element with address of l for (x=0; x<4; x++) printf ("\n %d",*(arr[x])); }</pre>	<pre>1 2 3 4</pre>

Here, arr contains the addresses of int variables i, j, k and l. The for loop is used to print the values present at these addresses.

A two-dimensional array can be represented using pointer to an array. But, a two-dimensional array can be expressed in terms of array of pointers also.

Using array of pointers a two dimensional array can be defined as,

data type *arr_ptr [size];

where data type refers to the data type of the array. arr_ptr refers to the name of the array and size is the maximum number of elements in the row.

Example `int *arr [3];`

Here, p is an array of pointers, and then following notations are used to refer elements.

p [i] --> points the address of the element ith row,

p[i] +j --> points the address of the element ith row and jth column

***(p[i] +j) --> value at ith row and jth column.**

Array of pointers and Strings

- ❖ Each element of the array is a pointer to a data type (in this case character).
- ❖ A block of memory (probably in the constant area) is initialized for the array elements.
- ❖ Declaration:

char *names [10]; // array of 10 character pointers.

- ❖ In this declaration names [] is an array of pointers. It contains base address of respective names. That is, base address of first name is stored in name [0] etc., a character pointer etc.
- ❖ The main reason to store array of pointers is to make more efficient use of available memory.

```
# include <stdio.h>
int main ()
{
    char *name [] = {
        "Illegal month",
        "January", "February", "March",
        "April", "May", "June",
        "July", "August", "September",
        "October", "November", "December"
    };
}
```

The pointers are initialized like so



Note: When we are using an array of pointers to strings we can initialize the string at the place where we are declaring the array, but we cannot receive the string from keyword using scanf ().

Review Questions:

4. Write a C program to read in an array of integers. Instead of using subscripting, however, employ an integer pointer that points to the elements currently being read in and which is incremented each time.
5. Write a program to reverse the strings stored in array of pointers.
6. Write a program to find rank of a matrix using pointers.
7. Write a C program using pointers to read in an array of integers and print its elements in reverse order.
8. Write a 'C' Program to compute the sum of all elements stored in an array using pointers.
9. Write a C program to find the desired element in an array of N elements. Use pointers to search an element.
10. a) Explain how strings can be stored using a multidimensional arrays?
b) What are the string in-built functions available? Write in detail about each one of them with an example.
11. What is the purpose of array of pointers? illustrate with an example.
12. Distinguish between array of pointers and pointer to array with examples.

Snippet Programs:

1.

```
#include<stdio.h>
int main()
{
    int arr[2][2][2] = {10, 2, 3, 4, 5, 6, 7, 8};
    int *p, *q;
    p = &arr[1][1][1];
    q = (int*) arr;
    printf("%d, %d\n", *p, *q);
    return 0;
}
```
2.

```
#include<stdio.h>
int main()
{
    int a[3][4] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12 };
    printf("%u, %u, %u\n", a[0]+1, *(a[0]+1), *(*a+0)+1));
    return 0;
}
```

```
}
```

```
3. #include<stdio.h>
int main()
{
    int arr[3] = {2, 3, 4};
    char *p;
    p = arr;
    p = (char*)((int*)(p));
    printf("%d, ", *p);
    p = (int*)(p+1);
    printf("%d", *p);
    return 0;
}
```

Dynamic Memory Allocation

- ✓ The exact size of array is unknown until the compile time, i.e., time when a compiler compiles code written in a programming language into an executable form. The size of array you have declared initially can be sometimes insufficient and sometimes more than required.
- ✓ Dynamic memory allocation allows a program to obtain more memory space, while running or to release space when no space is required.
- ✓ Although, C language inherently does not have any technique to allocate memory dynamically, there are 4 library functions under "stdlib.h" for dynamic memory allocation.

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

malloc()

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

Syntax of malloc()

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

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

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

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

calloc()

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

Syntax of calloc()

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

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

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

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

Differences between malloc() and Calloc():

S.no	malloc	calloc
1	It allocates only single block of requested memory	It allocates multiple blocks of requested memory
2	<pre>int *ptr;ptr = malloc(20 * sizeof(int));</pre> For the above, 20*4 bytes of memory only allocated in one block. Total = 80 bytes	<pre>int *ptr;Ptr = calloc(20, 20 * sizeof(int));</pre> For the above, 20 blocks of memory will be created and each contains 20*4 bytes of memory. Total = 1600 bytes
3	malloc () doesn't initialize the allocated memory. It contains garbage values	calloc () initializes the allocated memory to zero
4	type cast must be done since this function returns void pointer <pre>int *ptr;ptr = (int*)malloc(sizeof(int)*20);</pre>	Same as malloc () function <pre>int *ptr;ptr = (int*)calloc(20, 20 * sizeof(int));</pre>

free()

Dynamically allocated memory with either calloc() or malloc() does not get return on its own. The programmer must use free() explicitly to release space.

syntax of free()

```
free(ptr);
```

This statement causes the space in memory pointer by ptr to be deallocated.

Example 1:

```
1.      #include <stdio.h>
2.      #include <stdlib.h>
3.      int main()
4.      {
5.      int n,i,*ptr,sum=0;
6.      printf("Enter number of elements: ");
7.      scanf("%d",&n);
8.      ptr=(int*)malloc(n*sizeof(int));
9.      if(ptr==NULL)
10.     {
11.     printf("Error! memory not allocated.");
12.     exit(0);
13.     }
14.     printf("Enter elements of array: ");
15.     for(i=0;i<n;++i)
16.     {
17.     scanf("%d",ptr+i);
18.     sum+=*(ptr+i);
19.     }
20.     printf("Sum=%d",sum);
21.     free(ptr);
22.     return 0;
23.     }
```

- The above Program finds sum of n elements entered by user. To perform this program, allocate memory dynamically using malloc() function.
- In line 8, a size of (n*4) bytes will be allocated using malloc() and the address of first byte is stored in ptr. If the allocation is not successful, malloc returns NULL.
- Line numbers 15-20 finds and displays the sum of entered values.
- In Line 21, free() deallocates the memory allocated by malloc().

Example 2:

In the above program if line 8 is replaced by the following statement

ptr=(int*)calloc(n,sizeof(int));

then, calloc() allocates contiguous space in memory for an array of n elements each of size of int, i.e, 4 bytes.

realloc()

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

Syntax of realloc()

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

Here, *ptr* is reallocated with size of newsiz.

```
1.      #include <stdio.h>
2.      #include <stdlib.h>
3.      int main()
4.      {
5.      int *ptr,i,n1,n2;
6.      printf("Enter size of array: ");
7.      scanf("%d",&n1);
8.      ptr=(int*)malloc(n1*sizeof(int));
9.      printf("Address of previously allocated memory: ");
10.     for(i=0;i<n1;++i)
11.     printf("%u\t",ptr+i);
12.     printf("\nEnter new size of array: ");
13.     scanf("%d",&n2);
14.     ptr=realloc(ptr,n2);
15.     for(i=0;i<n2;++i)
16.     printf("%u\t",ptr+i);
17.     return 0;
18.     }
```

In the above Program,

- In line 8, a size of (n1*4) bytes will be allocated using malloc() and the address of first byte is stored in ptr. Line numbers 15-20 finds and displays the sum of entered values.
- In Line 14, The allocated memory is modified into(n2*4) bytes using realloc() function.

Review Questions:

1. Explain how memory management can be done dynamically in C language?
2. Explain the differences between memory allocations malloc() and calloc()

Snippet Programs:

1.

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

int main()
{
    int *p;
    p = (int *)malloc(20);
    printf("%d\n", sizeof(p));
    free(p);
    return 0;
}
```
2.

```
#include<stdio.h>
#include<stdlib.h>
#define MAXROW 3
#define MAXCOL 4

int main()
{
    int (*p)[MAXCOL];
    p = (int (*) [MAXCOL])malloc(MAXROW *sizeof(*p));
    return 0;
}
```
3.

```
#include<stdio.h>
#include<stdlib.h>
int main()
{
    int *p;
    p = (int *)malloc(256 * 256);
    if(p == NULL)
        printf("Allocation failed");
    return 0;
}
```

3.14 Command Line Arguments

- ✓ It is possible to pass some values from the command line to your C programs when they are executed. These values are called **command line arguments** and many times they are important for your program specially when you want to control your program from outside instead of hard coding those values inside the code.
- ✓ The command line arguments are handled using `main()` function arguments where **argc**(argument count) refers to the number of arguments passed, and **argv**[argument vector] is a pointer array which points to each argument passed to the program. Following is a simple example which checks if there is any argument supplied from the command line and take action accordingly:

Enter the following code and compile it:

```
#include <stdio.h>

int main(int argc, char *argv[])
{
    int x;

    printf("%d\n",argc);
    for (x=0; x<argc; x++)
        printf("%s\n",argv[x]);
    return 0;
}
```

In this code, the main program accepts two parameters, `argv` and `argc`. The `argv` parameter is an array of pointers to string that contains the parameters entered when the program was invoked at the UNIX command line. The `argc` integer contains a count of the number of parameters. This particular piece of code types out the command line parameters. To try this, compile the code to an executable file named **aaa** and type **aaa xxx yy zzz**. The code will print the command line parameters `xxx`, `yy` and `zzz`, one per line.

The **`char *argv[]`** line is an array of pointers to string. In other words, each element of the array is a pointer, and each pointer points to a string (technically, to the first character of the string). Thus, **`argv[0]`** points to a string that contains the first parameter on the command line (the program's name), **`argv[1]`** points to the next parameter, and so on. The `argc` variable tells you how many of the pointers in the array are valid. You will find that the preceding code does nothing more than print each of the valid strings pointed to by `argv`.

Because `argv` exists, you can let your program react to command line parameters entered by the user fairly easily. For example, you might

have your program detect the word **help** as the first parameter following the program name, and dump a help file to stdout. File names can also be passed in and used in your fopen statements.

```
#include <stdio.h>

int main( int argc, char *argv[] )
{
    if( argc == 2 )
    {
        printf("The argument supplied is %s\n", argv[1]);
    }
    else if( argc > 2 )
    {
        printf("Too many arguments supplied.\n");
    }
    else
    {
        printf("One argument expected.\n");
    }
}
```

When the above code is compiled and executed with a single argument, it produces the following result.

```
./a.out testing
The argument supplied is testing
```

When the above code is compiled and executed with a two arguments, it produces the following result.

```
./a.out testing1 testing2
Too many arguments supplied.
```

When the above code is compiled and executed without passing any argument, it produces the following result.

```
./a.out
One argument expected
```

It should be noted that **argv[0]** holds the name of the program itself and **argv[1]** is a pointer to the first command line argument supplied, and *argv[n] is the last argument. If no arguments are supplied, argc will be one, otherwise and if you pass one argument then **argc** is set at 2.

You pass all the command line arguments separated by a space, but if argument itself has a space then you can pass such arguments by putting them inside double quotes "" or single quotes '. Let us re-write above example once again where we will print program name and we also pass a command line argument by putting inside double quotes:

```
#include <stdio.h>

int main( int argc, char *argv[] )
{
    printf("Program name %s\n", argv[0]);

    if( argc == 2 )
    {
        printf("The argument supplied is %s\n", argv[1]);
    }
    else if( argc > 2 )
    {
        printf("Too many arguments supplied.\n");
    }
    else
    {
        printf("One argument expected.\n");
    }
}
```

When the above code is compiled and executed with a single argument separated by space but inside double quotes, it produces the following result.

```
$/a.out "testing1 testing2"
```

Program name ./a.out

The argument supplied is testing1 testing2

The following example program demonstrates:

```
#include <stdio.h>

void main( int argc, char *argv[] )
{
    int ctr;
    for( ctr=0; ctr < argc; ctr++ )
    {
        puts( argv[ctr] );
    }
}
```

```
}
```

If this program is run from the command line as follows:

```
$/a.out stooges moe larry curley
```

-- the output is:

```
stooges
moe
larry
curley
```

Parameter Parsing Example

Command-line parameters can be used for many things. Let's take a look at another example that will print different things depending on the flags given. If the `-f` flag is used then the given argument is printed once. If the `-d` flag is used the argument is printed twice.

Let's look at the example:

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

void usage(void)
{
    printf("Usage:\n");
    printf(" -f<name>\n");
    printf(" -d<name>\n");
    exit (8);
}

int main(int argc, char *argv[])
{
    printf("Program name: %s\n", argv[0]);

    while ((argc > 1) && (argv[1][0] == '-'))
    {
        switch (argv[1][1])
        {
            case 'f':
                printf("%s\n", &argv[1][2]);
                break;
        }
    }
}
```

```

        case 'd':
            printf("%s\n",&argv[1][2]);
            printf("%s\n",&argv[1][2]);
            break;

        default:
            printf("Wrong Argument: %s\n", argv[1]);
            usage();
    }

    ++argv;
    --argc;
}
return (0);
}

```

After compiling you can run the program with the following parameters:

- # program
- # program -fhello
- # program -dbye
- # program -fhello -dbye
- # program -w

Note: that there is no space between the flag and the flag argument.

First the program name is printed (it will always be printed.)

The “while loop” looks for the dash sign. In case of the -f the flag argument is printed once. If the flag is -d then the flag argument is printed twice.

The argv[][] array will be used as follows:

For example: -f<name>

- argv[1][0] contains the dash sign
- argv[1][1] contains the “f”
- argv[1][2] contains the flag argument name

If a wrong flag sign is given then the usage is printed onto the screen.

Review Questions

1. Define Command line arguments? Does C support Command line arguments? Justify your answer.
2. Explain Commandline arguments in C language through an example.

Snippet Programs:

1.

```
#include<stdio.h>
int main(int argc, char **argv)
{
    printf("%c\n", **++argv);
    return 0;
}

#include<stdio.h>
#include<stdlib.h>
int main(int argc, char **argv)
{
    printf("%s\n", *++argv);
    return 0;
}
```
2.

```
#include<stdio.h>
void fun(int);
int main(int argc)
{
    printf("%d ", argc);
    fun(argc);
    return 0;
}
void fun(int i)
{
    if(i!=4)
        main(++i);
}
```
3.

```
#include<stdio.h>
int main(int argc, char *argv[])
{
    while(--argc>0)
        printf("%s", *++argv);
    return 0;
}
```

By :

Mr.Shaik Masthan Valli
Asst.,Prof..
Dept of CSE
GITAM University
HTP Campus.

Enumerated Types (enum) in C

In C programming language, an enumeration is used to create user-defined datatypes. Using enumeration, integral constants are assigned with names and we use these names in the program. Using names in programming makes it more readable and easy to maintain.

Enumeration is the process of creating user defined datatype by assigning names to integral constants

We use the keyword **enum** to create enumerated datatype. The general syntax of enum is as follows...

```
enum {name1, name2, name3, ... }
```

In the above syntax, integral constant '0' is assigned to name1, integral constant '1' is assigned to name2 and so on. We can also assign our own integral constants as follows...

```
enum {name1 = 10, name2 = 30, name3 = 15, ... }
```

In the above syntax, integral constant '10' is assigned to name1, integral constant '30' is assigned to name2 and so on.

Example Program for enum with default values

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

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

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

```
enum day { Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday} ;
```

```
void main(){
```

```
    enum day today;
```

```
    today = tuesday ;
```

```
    printf("\ntoday = %d ", today) ;
```

```
}
```

In the above example program a user defined datatype "**day**" is created with seven values, Monday as integral constant '0', Tuesday as integral constant '1', Wednesday as integral constant '2', Thursday as integral constant '3', Friday as integral constant '4', Saturday as integral constant '5', and Sunday as integral constant '6'.

and sunday as integral constant '6'. Here, when we display tuesday it displays the respective integral constant '1'.

We can also change the order of integral constants, consider the following example program.

Example Program for enum with changed integral constant values

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

enum day { Monday = 1, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday} ;

void main(){

    enum day today;

    today = tuesday ;

    printf("\ntoday = %d ", today) ;

}
```

In the above example program, the integral constant value starts with '1' instead of '0'. Here, tuesday value is displayed as '2'.

We can also create enum with our own integral constants, consider the following example program.

Example Program for enum with defferent integral constant values

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

enum threshold {low = 40, normal = 60, high = 100} ;

void main(){

    enum threshold status;

    status = low ;

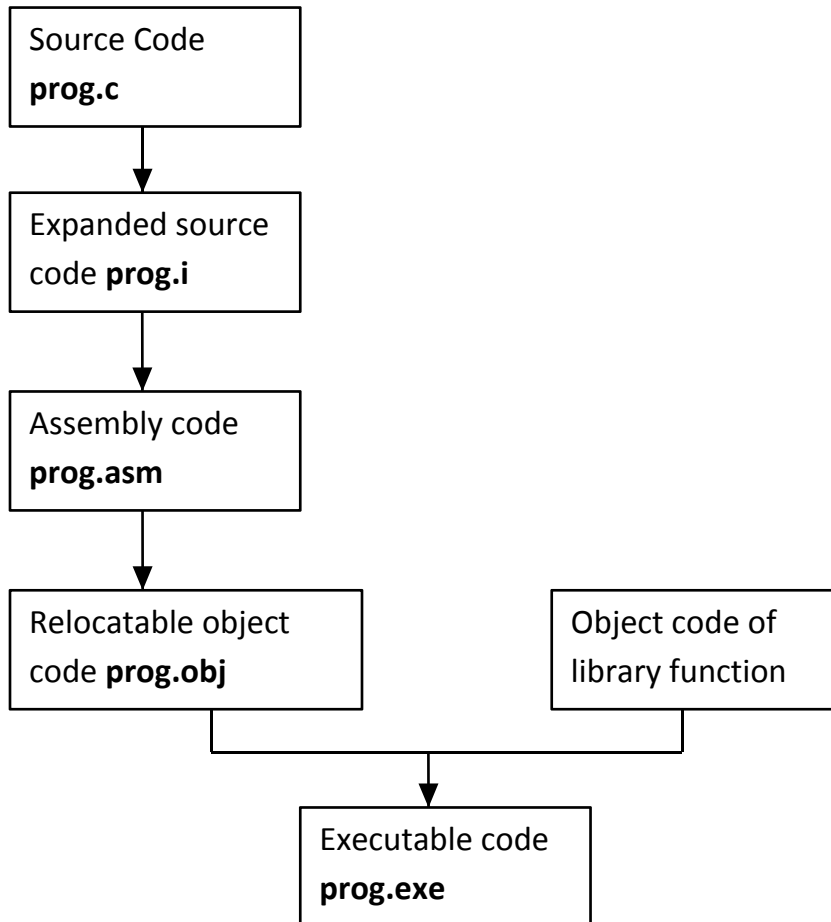
    printf("\nYou are at %d state. Work hard to improve!!", status) ;

}
```

Some times we may assign our own integral constant from other than the first name. In this case, compiler follows default integral constants for the name that is before the user-defined integral constant and from user-defined constant onwards it resumes. Consider the following example program.

C PREPROCESSOR

- Preprocessor is exactly what its name implies.
- It is a program that process the source program before it is passed to the compiler.
- Preprocessor contains certain commands in it, to process the source code. These commands are known as “**DIRECTIVE**”.
- All these commands together, can be considered as a language with in c- language.
- Without having knowledge about preprocessor , all c- programmers depend on it. **This feature does not exist in other higher level languages.**



Each of the preprocessor directive begins with a # symbol.

There directives can be placed any where in the program, but generally these are written in the beginning of the program, before main() or before a particular function.

Some of the preprocessor directives are:

1. #include directive (file inclusion)
2. #define directive (macro substitution)
3. #undef directive
4. Conditional compilation directives

i) **#define directive:**

```
#define RANGE 10
main()
{
    int k;
    for(k=1;k<=RANGE; k++)
    {
        printf("k=%d\n",k);
    }
}
```

Value is constant throughout the program.

In the above program, instead of writing 10, in for loop we are writing as RANGE

RANGE is also defined before main() using statement.

#define RANGE 10

The above statement is called as **MACRO DEFINATION**

During preprocessing, preprocessor replaces, each occurrence of word RANGE with 10.

```
#define A 25
main()
{
    int i,j;
    printf("enter i value");
    scanf("%d",&i);
    j=A*i;
    printf("product=%d",j);
}
```

Here ; Range & A are called as “Macro templates”.

10 & 25 are called as “Macro Expansions”.

When we compile the program, before source code passes to compiler, it is evaluated by preprocessor.

- Preprocessors checks for the Macro definitions.
- According to the Macro definition given, preprocessor searches the entire for “Macro templates”.
- Whenever, it finds these “Macro templates”, preprocessor replaces them with “Macro expansions”.

- After completion of this procedure, program will be sent to compiler.

```
#define VALUE 3+5
main()
{
  int i;
  clrscr();
  i=VALUE * VALUE;
  printf("%d",i);
}
```

i=3+5*3+5
i=3+15+5
i=23

Reason: * has priority over +

First '*' is performed, and then '+' is performed.

NOTE:

- Generally, capital letters are used for writing Macro templates. (Small letters can also be used)
- Macro template & Macro expansion are separated by space.

```
#define RANGE 10
      ↓      ↓
      M.T    M.E
```

MACROS WITH ARGUMENTS:

Macros can have arguments, just like functions

```
#define AREA(x) (3.14*x*x)
main()
{
  float i1=2.2,i2=3.4,j;
  clrscr();
  j=AREA(i1);
  printf("Area=%f",j);
  j=AREA(i2);
  printf("Area=%f",j);
}
```

Preprocessors will replace AREA(x) with 3.14*x*x

Also i1&i2 will be substituted, based on function calls

J=AREA(i1) -> when this function call is executed, i1 is assigned to x.

J=AREA(i2) ->i2 will be assigned to x.

Difference between Macro and Functions:

In a Macro call, preprocessor replaces Macro template with its Macro expansion. Macro won't return any value to main().

In a function call, the control is passed to a function, with certain arguments and some value will be returned back to the calling function.

Macros makes programs to run faster, but increases the size of program, whereas, functions decreases the size of program.

```
#define AREA(x) (3.14*x*x)
main()
{
float i=2.2,t;
clrscr();
t=AREA(i);
printf("Area=%f",t);
}
```

1. Preprocessor will replace AREA(x) with 3.14*x*x
2. When function call AREA(i) is made, then i is assigned to x and result is calculated as 3.14*i*i

ii) File Inclusion:

- It is a process of inserting external files, containing functions into the c-program
- This avoids re-writing those functions in the program.

Suppose we have to use same function in 5 diff programs, then file inclusion technique can be used.

An external file can be loaded into the c- program using #include directive.

General form of file inclusion is:

```
#include filename
```

#include → preprocessor directive for file inclusion

Filename → name of the file containing the required function define to be included in a c-program.

If filename is declared as: #include<filename>

then, the specified file is searched only in standard directories.

If filename is declared as: #include "filename"

then, the specified file is first searched in the current directories and then in the standard directories.

iii) **#undef Directive**

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

#undef macro-template

can be used.

EXAMPLE: #define PI 3.1414

```
Void main()
{
float a,r=3;
clrscr();
a=PI*3*3;
printf("AREA = %d",a);
#undef PI
a=PI*3*3; /* Gives error undefined PI*/
printf("Area =%d",a);
getch();
}
```

For this program it gives error because the macro PI is undefined. After undefining macro we are using that macro in the program therefore it gives error.

iv) **Conditional Compilation:**

It is a process of selecting a particular segment of code for compilation based on condition.

```
main()
{
#define one
#ifdef One
printf("hi");
printf("hellow");
#else
printf("c-lab");
#endif
printf("welcome");
}
```

Output:

- i) hi hellow welcome
- ii) c-lab welcome (if #define is commented)

- c- preprocessor provides a compilation directive which is used to select alternate segment of code depending upon the condition.
- If there are 2 different versions of a program, it will take more memory to store the program. This problem can be solved by including both version in a single program.
- Then a particular version can be selected based on the requirement.
- In large programs, generally programmers writes comments for each and every statement. If we use comment, instead of conditional compilation directives, we would end up with nested comments. Sometimes, it is not possible to write nested comments in C.

Function Returning Non-integers:

C function returns a value of type 'int' as a default value, when no return type is specified.

Two steps must be performed ,to enables a calling function, to receive a non-integer value from the called function:

1. Return-type of the function should be specified in the function header.
2. The called function must be declared at the start of body, in the calling function, like any other variable. This is to tell the calling function, the type of data, that the function is actually returning,.

```
float m(float x, float y);  
main()  
{  
  float a,b, mul();  
  a=12.34;  
  b=7.32;  
  printf("%f", mul(a,b));  
}  
float mul(float x, float y)  
{  
  return x*y;  
}
```

Files in C

Generally, a file is used to store user data in a computer. In other words, computer stores the data using files. we can define a file as follows...

File is a collection of data that stored on secondary memory like haddisk of a computer.

C programming language supports two types of files and they are as follows...

- **Text Files (or) ASCII Files**
- **Binary Files**

Text File (or) ASCII File - The file that contains ASCII codes of data like digits, alphabets and symbols is called text file (or) ASCII file.

Binary File - The file that contains data in the form of bytes (0's and 1's) is called as binary file. Generally, the binary files are compiled version of text files.

File Operations in C

The following are the operations performed on files in c programming language...

- **Creating (or) Opening a file**
- **Reading data from a file**
- **Writing data into a file**
- **Closing a file**

All the above operations are performed using file handling functions available in C.

File Handling Functions in C

File is a collection of data that stored on secondary memory like hard disk of a computer.

The following are the operations performed on files in the c programming language...

- **Creating (or) Opening a file**
- **Reading data from a file**
- **Writing data into a file**
- **Closing a file**

All the above operations are performed using file handling functions available in C.

Creating (or) Opening a file

To create a new file or open an existing file, we need to create a file pointer of FILE type.

Following is the sample code for creating file pointer.

```
File *f_ptr ;
```

We use the pre-defined method **fopen()** to create a new file or to open an existing file. There are different modes in which a file can be opened. Consider the following code...

```
File *f_ptr ;
```

```
*f_ptr = fopen("abc.txt", "w") ;
```

The above example code creates a new file called **abc.txt** if it does not exist otherwise it is opened in writing mode.

In C programming language, there are different modes available to open a file and they are shown in the following table.

S. No.	Mode	Description
1	r	Opens a text file in reading mode.
2	w	Opens a text file in writing mode.
3	a	Opens a text file in append mode.
4	r+	Opens a text file in both reading and writing mode.
5	w+	Opens a text file in both reading and writing mode. It sets the cursor position to the beginning of the file if it exists.
6	a+	Opens a text file in both reading and writing mode. The reading operation is performed from the beginning and writing operation is performed at the end of the file.

Note - The above modes are used with text files only. If we want to work with binary files we use

rb, wb, ab, rb+, wb+ and ab+.

Reading from a file

The reading from a file operation is performed using the following pre-defined file handling methods.

1. `getc()`
2. `getw()`
3. `fscanf()`
4. `fgets()`
5. `fread()`

- **`getc(*file_pointer)`** - This function is used to read a character from specified file which is opened in reading mode. It reads from the current position of the cursor. After reading the character the cursor will be at next character.

Example Program to illustrate `getc()` in C.

```
#include<stdio.h>
#include<conio.h>
int main(){
    FILE *fp;
    char ch;
    clrscr();
    fp = fopen("MySample.txt","r");
    printf("Reading character from the file: %c\n",getc(fp));
    ch = getc(fp);
    printf("ch = %c", ch);
    fclose(fp);
    getch();
    return 0;
}
```

- **`getw(*file_pointer)`** - This function is used to read an integer value form the specified file which is opened in reading mode. If the data in file is set of characters then it reads ASCII values of those characters.

```
#include<stdio.h>
#include<conio.h>
int main(){
    FILE *fp;
    int i,j;
```

```

clrscr();
fp = fopen("MySample.txt","w");
putw(65,fp); // inserts A
putw(97,fp); // inserts a
fclose(fp);
fp = fopen("MySample.txt","r");
i = getw(fp); // reads 65 - ASCII value of A
j = getw(fp); // reads 97 - ASCII value of a
printf("SUM of the integer values stored in file = %d", i+j); // 65 + 97 = 162
fclose(fp);
getch();
return 0;
}

```

- ***fscanf(*file_pointer, typeSpecifier, &variableName)*** - This function is used to read multiple datatype values from specified file which is opened in reading mode.

```

#include<stdio.h>
#include<conio.h>
int main(){
    char str1[10], str2[10], str3[10];
    int year;
    FILE * fp;
    clrscr();
    fp = fopen ("file.txt", "w+");
    fputs("We are in 2016", fp);
    rewind(fp); // moves the cursor to beginning of the file
    fscanf(fp, "%s %s %s %d", str1, str2, str3, &year);
    printf("Read String1 - %s\n", str1 );
    printf("Read String2 - %s\n", str2 );
    printf("Read String3 - %s\n", str3 );
    printf("Read Integer - %d", year );
    fclose(fp);
    getch();

    return
}

```

- ***fgets(variableName, numberOfCharacters, *file_pointer)*** - This method is used for reading a set of characters from a file which is opened in reading mode starting from the current cursor position. The fgets() function reading terminates with reading NULL character.

```
#include<stdio.h>
#include<conio.h>
int main(){
    FILE *fp;
    char *str;
    clrscr();
    fp = fopen ("file.txt", "r");
    fgets(str,6,fp);
    printf("str = %s", str);
    fclose(fp);
    getch();
    return 0;
}
```

- ***fread(source, sizeofReadingElement, numberOfCharacters, FILE *pointer)*** - This function is used to read specific number of sequence of characters from the specified file which is opened in reading mode.

```
#include<stdio.h>
#include<conio.h>
int main(){
    FILE *fp;
    char *str;
    clrscr();
    fp = fopen ("file.txt", "r");
    fread(str,sizeof(char),5,fp);
    str[strlen(str)+1] = 0;
    printf("str = %s", str);
    fclose(fp);
    getch();
    return 0;
}
```

Writing into a file

The writing into a file operation is performed using the following pre-defined file handling methods.

1. **putc()**
2. **putw()**
3. **fprintf()**
4. **fputs()**
5. **fwrite()**

- ***putc(char, *file_pointer)*** - This function is used to write/insert a character to the specified file when the file is opened in writing mode.

```

#include<stdio.h>
#include<conio.h>
int main(){
    FILE *fp;
    char ch;
    clrscr();
    fp = fopen("C:/TC/EXAMPLES/MySample.txt","w");
    putc('A',fp);
    ch = 'B';
    putc(ch,fp);
    fclose(fp);
    getch();
    return 0;
}

```

- ***putw(int, *file_pointer)*** - This function is used to writes/inserts an integer value to the specified file when the file is opened in writing mode.

```

#include<stdio.h>
#include<conio.h>
int main(){
    FILE *fp;
    int i;
    clrscr();
    fp = fopen("MySample.txt","w");
    putw(66,fp);
    i = 100;
    putw(i,fp);
    fclose(fp);
    getch();
    return 0;
}

```

- ***fprintf(*file_pointer, "text")*** - This function is used to writes/inserts multiple lines of text with mixed data types (char, int, float, double) into specified file which is opened in writing mode.

```

#include<stdio.h>
#include<conio.h>
int main(){
    FILE *fp;
    char *text = "\nthis is example text";
    int i = 10;
    clrscr();
    fp = fopen("MySample.txt","w");
    fprintf(fp,"This is line1\nThis is line2\n%d", i);
    fprintf(fp,text);
    fclose(fp);
    getch();
    return 0;
}

```

- ***fputs("string", *file_pointer)*** - This method is used to insert string data into specified file which is opened in writing mode.

```
#include<stdio.h>
#include<conio.h>
int main(){
    FILE *fp;
    char *text = "\nthis is example text";
    clrscr();
    fp = fopen("MySample.txt","w");
    fputs("Hi!\nHow are you?",fp);
    fclose(fp);
    getch();
    return 0;

}
```

- ***fwrite("StringData", sizeof(char), numberOfCharacters, FILE *pointer)*** - This function is used to insert specified number of characters into a binary file which is opened in writing mode.

```
#include<stdio.h>
#include<conio.h>
int main(){
    FILE *fp;
    char *text = "Welcome to C Language";
    clrscr();
    fp = fopen("MySample.txt","wb");
    fwrite(text,sizeof(char),5,fp);
    fclose(fp);
    getch();
    return 0;

}
```

Closing a file

Closing a file is performed using a pre-defined method `fclose()`.

```
fclose( *f_ptr )
```

The method `fclose()` returns '0' on success of file close otherwise it returns EOF (End Of File).

Cursor Positioning Functions in Files

C programming language provides various pre-defined methods to set the cursor position in files. The following are the methods available in c, to position cursor in a file.

1. **`ftell()`**
2. **`rewind()`**
3. **`fseek()`**

- ***ftell(*file_pointer)*** - This function returns the current position of the cursor in the file.

Example Program to illustrate ftell() in C.

```
#include<stdio.h>
#include<conio.h>
int main(){
    FILE *fp;
    int position;
    clrscr();
    fp = fopen ("file.txt", "r");
    position = ftell(fp);
    printf("Cursor position = %d\n",position);
    fseek(fp,5,0);
    position = ftell(fp);
    printf("Cursor position = %d", position);
    fclose(fp);
    getch();
    return 0;
}
```

rewind(*file_pointer) - This function is used reset the cursor position to the beginning of the file.

Example Program to illustrate rewind() in C.

```
#include<stdio.h>
#include<conio.h>
int main(){
    FILE *fp;
    int position;
    clrscr();
    fp = fopen ("file.txt", "r");
    position = ftell(fp);
    printf("Cursor position = %d\n",position);
    fseek(fp,5,0);
    position = ftell(fp);
```

```

printf("Cursor position = %d\n", position);
rewind(fp);
position = ftell(fp);
printf("Cursor position = %d", position);
fclose(fp);
getch();
return 0;
}

```

fseek(*file_pointer, numberOfCharacters, fromPosition) - This function is used to set the cursor position to the specific position. Using this function we can set the cursor position from three different position they are as follows.

- from beginning of the file (indicated with 0)
- from current cursor position (indicated with 1)
- from ending of the file (indicated with 2)

Example Program to illustrate fseek() in C.

```

#include<stdio.h>
#include<conio.h>
int main(){
    FILE *fp;
    int position;
    clrscr();
    fp = fopen ("file.txt", "r");
    position = ftell(fp);
    printf("Cursor position = %d\n",position);
    fseek(fp,5,0);
    position = ftell(fp);
    printf("Cursor position = %d\n", position);
    fseek(fp, -5, 2);
    position = ftell(fp);
    printf("Cursor position = %d", position);
    fclose(fp);
    getch();

    return 0;
}

```

FUNCTIONS

Definition : A function is self contained block of statements that performs a particular task.

C -functions are divided into two parts

- 1) Pre-defined functions(library functions)
- 1) User-defined functions

library functions OR pre-defined functions are not required to be written by us where as user define function has to be developed by the user at the time of writing program. Ex: printf(), scanf(), sqrt(), pow() etc.,,,

To make use of user defined function, we need to write three elements that are related to function.

- 1) Function declaration
- 1) Function definition
- 1) Function call

Function declaration / signature / prototype :

Just like variables , functions are also to be declared before we use them in the program. A function declaration consists of four parts.

- 1) Function type(return type)
- 1) Function name
- 1) Parameter List
- 1) Terminating semicolon.

The general format of declaring a function is :

return_type function_name(parameter list);

Example: void display();
 int square(int m);
 int mul(int m, int n);

A prototype declaration may be placed in two places in a program

- 1) Above all the functions (including main)
 - 2) Inside a function definition.
- When we place the declaration above all the functions (in the global declaration section), the prototype is referred to as a global prototype. Such declarations are available for all the functions in the program.
 - When we place it in a function definition (in the local declaration section), this prototype is called local prototype.

Function call : is done inside main(). Whenever function call is made control from main() will be transferred to the definition of that function. After executing all the statements in the function control comes back to main().

General form of function call : **function_name() ;**

If arguments are being passed then : **function_name(arguments);**

Function definition : (also known as FUNCTION IMPLEMENTATION)

format of function definition is :-

```
return_type  fuction_name(arguments)
{
    Local variable declarations;
    stmt 1;
    stmt 2;
    -----
    -----
    return statement;
}
```

return type specifies the type of value that the function is going to return. Various return types arevoid, int, char, float, double, long int etc.,,

function name is the name of function. It is given the user.

- A FUNCTION IN WHICH FUNCTION CALL IS MADE IS KNOWN AS CALLING FUNCTION.
- A FUNCTION WHICH IS CALLED BY OTHER FUNCTION IS KNOWN AS CALLED FUNCTION.

Example :

```
main ()
{
    printf("hello");
    display();
}
```

In above sample code **display()** is called function and **main()** is calling function.

main () is a **specially recognized user-defined function**. That is ,

- **Declaration** of main() is **pre-defined**
- **Definition** of main() is **user-defined**

Arguments :

The mechanism of passing values between calling function and called function is known as ARGUMENT or PARAMETER.

There are 2 types of arguments or parameters :

1. Actual arguments
2. Formal arguments

➔ Variables in function call are actual arguments.

➔ Variables in function definition are formal arguments.

The return statement can take one of the following forms

```
return;  
or  
return(expression);
```

The first one does not return any value; it acts much as closing brace of the function. When a return is encountered, the control is immediately passed back to the calling function.

An example of the use of a simple return is as follows

```
if(error)  
return;
```

The second form of the return with an expression returns the value of the expression. For example the function

```
int mul(int x,int y)  
{  
    int p;  
    p=x*y;  
    return(p);  
}
```

returns the value of the p which is the product of the values of x and y. The last two statements can be combined into one statement as follows.

```
return(x*y);
```

A function may have more than one return statement this situation arises when the value returned is based on certain conditions. For example

```
if(x<=0)  
return(0);
```

```
else  
    return(1);
```

TYPES OF FUNCTIONS BASED ON ARGUMENTS AND RETURN VALUES :

1. Functions without arguments and without return values
2. Functions with arguments and without return values
3. Functions with arguments and with return values
4. Functions without arguments and with return values

Functions without arguments and without return values

- No values will be passed from calling function to called function.
- No values will be returned to calling function from called function.

Example program :

```
#include<stdio.h>  
void add( ); // function declaration  
main() // calling function  
{  
    add(); // function call  
}  
  
void add( ) // function definition and called function  
{  
    int x,y,z;  
    printf("enter any two integers");  
    scanf("%d %d", &x,&y);  
    z=x+y;  
    printf("sum = %d",z);  
}
```

Functions with arguments and without return values

- (arguments) values will be passed from calling function to called function.
- No values will be returned to calling function from called function.

Example program :

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

```
void add( int a, int b ); // function declaration ; 'a' and 'b' are formal arguments
```

```
main() // calling function
```

```
{
```

```
    int x,y ;
```

```
    printf("enter any two integers");
```

```
    scanf("%d %d", &x,&y);
```

```
    add(x,y); // function call ; 'x' and 'y' are actual arguments
```

```
}
```

```
void add( int a, int b) // function definition and called function
```

```
{
```

```
    int c;
```

```
    c = a + b;
```

```
    printf("sum = %d",c );
```

```
}
```

Functions with arguments and with return values

- (arguments) values will be passed from calling function to called function.
- values will be returned to calling function from called function.

Example program :

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

```
int add(int a, int b ); // function declaration ; 'a' and 'b' are formal arguments
```

```
main() // calling function
```

```
{
```

```
    int x,y,z ;
```

```
    printf("enter any two integers");
```

```
    scanf("%d %d", &x,&y);
```

```
    z = add(x,y); // function call ; 'x' and 'y' are actual arguments
```

```
    printf("sum = %d",z);
```

```
}
```

```
int add( int x, int y) // function definition and called function
```

```
{
```

```
    int c;
```

```
    c = x+y;
```

```
    return c;
```

```
}
```

Functions without arguments and with return values

- No values will be passed from calling function to called function.
- values will be returned to calling function from called function.

Example program :

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

```
int add( ); // function declaration
```

```
main() // calling function
```

```
{
```

```
    int c;
```

```
    c = add( ); // function call
```

```
    printf("sum = %d",c);
```

```
}
```

```
int add( ) // function definition and called function
```

```
{
```

```
    int x,y,z ;
```

```
    printf("enter any two integers");
```

```
    scanf("%d %d", &x,&y);
```

```
    z = x+y;
```

```
    return z;
}
```

SCOPE OF A VARIABLE : is the extent to which it can be used in a program.

Based on this, variables are classified as LOCAL VARIABLES & GLOBAL VARIABLES.

- **Local variable** is restricted to a particular block in the program.
- **Global variable** is used in the entire program that is in all the functions present in the program.

Example :

```
#include<stdio.h>

int k=20; // global variable

void add ( );

main()
{
    int c =30; // 'c' is local variable to main( )
    add ( );
    printf("sum = %d", ( c+k ) ); // output is 50
}
```

```
void add( )  
  
{  
    int h = 40; // 'h' is local variable to add( )  
    printf(" addition = %d", ( h+k ) ); // output is 60  
}
```

RECURSION

It is a modular programming technique.

A function calling itself is known as recursive function i.e, function call statement for calling the same function appears inside the function.

In recursion, the same function acts as calling function and called function.

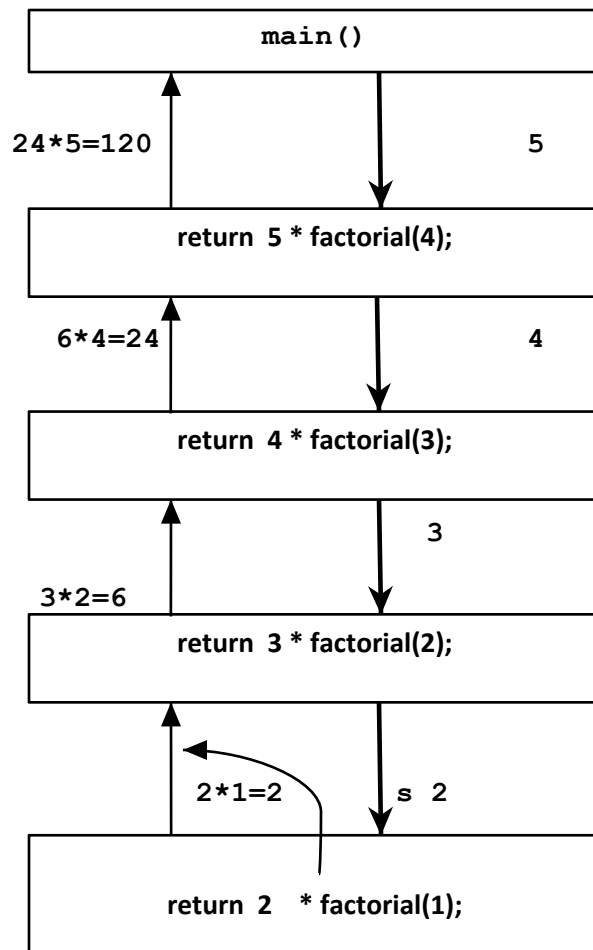
```
Ex:  int main()
    {
        Printf("here main() is a recursion function");
        main();
    }
```

There must be an exclusive terminating condition or else the execution will continue indefinitely.

Program to find factorial of number using recursion

```
#include<stdio.h>
int factorial( int k);
int main()
{
    int s,n;
    printf("enter a no for printing factorials from 1 to entered no. ");
    scanf("%d",&n);
    s = factorial(n);
    printf(" factorial = %d", s);
}
int factorial (int k )
{
    if ( k==1)
        return 1;
    else
        return k * factorial(k-1);
}
```

Consider $k = 5$



In the above program :-

- `main()` is calling function for factorial (5)
- `factorial (5)` is calling function for `factorial(4)`
- `factorial(4)` is calling function for `factorial(3)`
- `factorial(3)` is calling function for `factorial(2)`
- `factorial(2)` is calling function for `factorial(1)`

so, called function will return the value to its calling function.

Difference between recursion and iteration:

Both recursion and iteration are based on control structures.

- Iteration uses repetition structures to achieve looping and recursion uses selection structure to make repeated function calls.
- Both iteration and recursion use a condition to terminate. Iteration terminates when condition in loop fails. Recursion terminates when condition in 'if' statement becomes TRUE.
- Recursion helps us to solve a complex problem by breaking the program into smaller problems, that are similar to original problem. But each recursive call creates another copy of function in memory which consumes more and more memory. Thus, recursion should be applied only to larger programs.

Program to find " x power y " using recursion

```
#include<stdio.h>
int power(int,int);
int main()
{
    int b,e,res;
    printf("Enter the value of base and exponent");
    scanf("%d%d",&b,&e);
    res = power(b,e);
    printf("value of %d power %d is %d",b,e,res);
}

int power(int b, int e)
{
    if(e==0)
        return 1;
    else if(e==1)
        return b;
    else
        return (b * power(b,e-1));
}
```

Program to find fibonacci series using recursion

```
#include<stdio.h>
int main()
{
    int i,n;
    printf("Enter the n value:");
    scanf("%d",&n);
    printf("the fibonacci series is\n");
    for(i=0;i<n;i++)
    {
        printf("%d\n",fib(i));
    }
}
```

```
int fib(int n)
{
    if(n==0)
    {
        return(0);
    }
    if(n==1)
    {
        return(1);
    }
    else
    {
        return(fib(n-1)+fib(n-2));
    }
}
```

Program for binary search using recursion

```
#include <stdio.h>
main()
{
    int a[50];
    int n,no,x,result;

    printf("Enter the number of terms : ");
    scanf("%d",&no);

    printf("Enter the elements :\n");
    for(x=0;x<no; x++)
        scanf("%d",&a[x]);

    printf("Enter the number to be searched : ");
    scanf("%d",&n);

    result = binarysearch(a, n, 0, no-1); // function call

    if(result == -1)
        printf("Element not found");
    return 0;
}

binarysearch(int a[],int n, int low ,int high)
{
    int mid;
    if (low > high)
        return -1;

    mid = (low + high)/2;

    if(n == a[mid])
    {
        printf("The element is at position %d\n",mid+1);
        return 0;
    }
    if(n < a[mid])
    {
        high = mid - 1;
        binarysearch(a, n, low ,high); // recursive call
    }
    if(n > a[mid])
    {
        low = mid + 1;
        binarysearch(a, n, low ,high); // recursive call
    }
}
```

The concept of **dynamic memory allocation in c language** enables the C programmer to allocate memory at runtime. Dynamic memory allocation in c language is possible by 4 functions of stdlib.h header file.

1. malloc()
2. calloc()
3. realloc()
4. free()

Before learning above functions, let's understand the difference between static memory allocation and dynamic memory allocation.

static memory allocation	dynamic memory allocation
memory is allocated at compile time.	memory is allocated at run time.
memory can't be increased while executing program.	memory can be increased while executing program.
used in array.	used in linked list.

Now let's have a quick look at the methods used for dynamic memory allocation.

malloc()	allocates single block of requested memory.
calloc()	allocates multiple block of requested memory.
realloc()	reallocates the memory occupied by malloc() or calloc() functions.
free()	frees the dynamically allocated memory.

malloc() function in C

The malloc() function allocates single block of requested memory.

It doesn't initialize memory at execution time, so it has garbage value initially.

It returns NULL if memory is not sufficient.

The syntax of malloc() function is given below:

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

calloc() function in C

The `calloc()` function allocates multiple block of requested memory.

It initially initialize all bytes to zero.

It returns NULL if memory is not sufficient.

The syntax of `calloc()` function is given below:

1. `ptr=(cast-type*)calloc(number, byte-size)`

realloc() function in C

If memory is not sufficient for `malloc()` or `calloc()`, you can reallocate the memory by `realloc()` function. In short, it changes the memory size.

Let's see the syntax of `realloc()` function.

1. `ptr=realloc(ptr, new-size)`

free() function in C

The memory occupied by `malloc()` or `calloc()` functions must be released by calling `free()` function. Otherwise, it will consume memory until program exit.

Let's see the syntax of `free()` function.

- `free(ptr)`

Example on Malloc()

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

- **int** main(){
- **int** n,i,*ptr,sum=0;
- printf("Enter number of elements: ");
- scanf("%d",&n);
- ptr=(**int***)malloc(n***sizeof**(**int**)); //memory allocated using malloc
- **if**(ptr==NULL)
- {
- printf("Sorry! unable to allocate memory");
- exit(0);
- }
- printf("Enter elements of array: ");
- **for**(i=0;i<n;++i)
- {
- scanf("%d",ptr+i);
- sum+=*(ptr+i);
- }
- printf("Sum=%d",sum);
- free(ptr);
- **return** 0;
- }

Example on calloc()

- **#include**<stdio.h>
- **#include**<stdlib.h>
- **int** main(){
- **int** n,i,*ptr,sum=0;
- printf("Enter number of elements: ");
- scanf("%d",&n);
- ptr=(**int***)calloc(n,**sizeof**(**int**)); //memory allocated using calloc
- **if**(ptr==NULL)

```

• {
•     printf("Sorry! unable to allocate memory");
•     exit(0);
• }
• printf("Enter elements of array: ");
• for(i=0;i<n;++i)
• {
•     scanf("%d",ptr+i);
•     sum+=*(ptr+i);
• }
• printf("Sum=%d",sum);
• free(ptr);
• return 0;
• }

```

Command Line Arguments in C

The arguments passed from command line are called command line arguments. These arguments are handled by `main()` function.

To support command line argument, you need to change the structure of `main()` function as given below.

1. `int main(int argc, char *argv[])`

Here, **argc** counts the number of arguments. It counts the file name as the first argument.

The **argv[]** contains the total number of arguments. The first argument is the file name always.

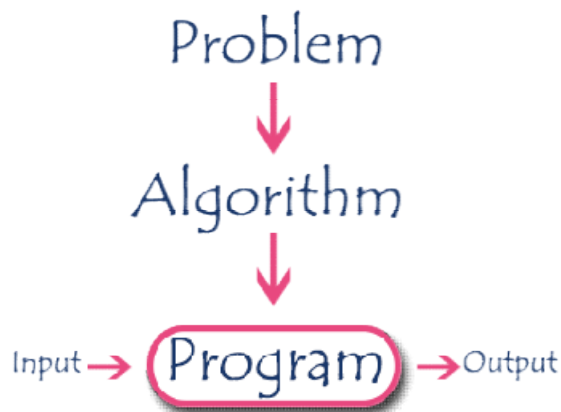
Introduction to Algorithms

What is an algorithm?

An algorithm is a step by step procedure to solve a problem. In normal language, algorithm is defined as a sequence of statements which are used to perform a task. In computer science, an algorithm can be defined as follows...

An algorithm is a sequence of unambiguous instructions used for solving a problem, which can be implemented (as a program) on a computer.

Algorithms are used to convert our problem solution into step by step statements. These statements can be converted into computer programming instructions which forms a program. This program is executed by computer to produce solution. Here, program takes required data as input, processes data according to the program instructions and finally produces result as shown in the following picture.



Specifications of Algorithms

Every algorithm must satisfy the following specifications...

1. **Input** - Every algorithm must take zero or more number of input values from external.
2. **Output** - Every algorithm must produce an output as result.
3. **Definiteness** - Every statement/instruction in an algorithm must be clear and unambiguous (only one interpretation).
4. **Finiteness** - For all different cases, the algorithm must produce result within a finite number of steps.
5. **Effectiveness** - Every instruction must be basic enough to be carried out and it also must be feasible.

Example for an Algorithm

Let us consider the following problem for finding the largest value in a given list of values.

Problem Statement : Find the largest number in the given list of numbers?

Input : A list of positive integer numbers. (List must contain at least one number).

Output : The largest number in the given list of positive integer numbers.

Consider the given list of numbers as 'L' (input), and the largest number as 'max' (Output).

Algorithm

1. **Step 1:** Define a variable 'max' and initialize with '0'.
2. **Step 2:** Compare first number (say 'x') in the list 'L' with 'max', if 'x' is larger than 'max', set 'max' to 'x'.
3. **Step 3:** Repeat step 2 for all numbers in the list 'L'.
4. **Step 4:** Display the value of 'max' as a result.

Write an algorithm to find all roots of a quadratic equation $ax^2+bx+c=0$.

Step 1: Start

Step 2: Declare variables a, b, c, D, x1, x2, rp and ip;

Step 3: Calculate discriminant

$$D \leftarrow b^2 - 4ac$$

Step 4: If $D \geq 0$

$$r1 \leftarrow (-b + \sqrt{D}) / 2a$$

$$r2 \leftarrow (-b - \sqrt{D}) / 2a$$

Display r1 and r2 as roots.

```
        Else
            Calculate real part and imaginary part
             $rp \leftarrow b/2a$ 
             $ip \leftarrow \sqrt{-D}/2a$ 
            Display  $rp+j(ip)$  and  $rp-j(ip)$  as roots
Step 5: Stop
5: Stop
```

Max-Min Problem

Let us consider a simple problem that can be solved by divide and conquer technique.

Problem Statement

The Max-Min Problem in algorithm analysis is finding the maximum and minimum value in an array.

Solution

To find the maximum and minimum numbers in a given array ***numbers[]*** of size ***n***, the following algorithm can be used. First we are representing the **naïve method** and then we will present **divide and conquer approach**.

Naïve Method

Naïve method is a basic method to solve any problem. In this method, the maximum and minimum number can be found separately. To find the maximum and minimum numbers, the following straightforward algorithm can be used.

```
Algorithm: Max-Min-Element (numbers[])
max := numbers[1]
min := numbers[1]

for i = 2 to n do
```

```

    if numbers[i] > max then
        max := numbers[i]
    if numbers[i] < min then
        min := numbers[i]
return (max, min)

```

Analysis

The number of comparison in Naive method is $2n - 2$.

The number of comparisons can be reduced using the divide and conquer approach. Following is the technique.

Divide and Conquer Approach

In this approach, the array is divided into two halves. Then using recursive approach maximum and minimum numbers in each halves are found. Later, return the maximum of two maxima of each half and the minimum of two minima of each half.

In this given problem, the number of elements in an array is $y - x + 1$, where y is greater than or equal to x .

$\text{Max-Min}(x, y)$ will return the maximum and minimum values of an array $\text{numbers}[x \dots y]$.

```

Algorithm: Max - Min(x, y)
if  $y - x \leq 1$  then
    return (max(numbers[x], numbers[y]), min((numbers[x], numbers[y])))
else
    (max1, min1) := maxmin(x,  $\lfloor (x + y) / 2 \rfloor$ )
    (max2, min2) := maxmin( $\lfloor (x + y) / 2 \rfloor + 1$ , y)
return (max(max1, max2), min(min1, min2))

```

Analysis

Let $T(n)$ be the number of comparisons made by $\text{Max-Min}(x, y)$, where the number of elements $n = y - x + 1$.

If $T(n)$ represents the numbers, then the recurrence relation can be represented as

$$T(n) = \begin{cases} 1 & \text{if } n=1 \\ 2T(n/2) + 2 & \text{if } n > 2 \end{cases}$$

Let us assume that n is in the form of power of 2. Hence, $n = 2^k$ where k is height of the recursion tree.

So,

$$T(n) = 2.T(n/2) + 2 = 2.(2.T(n/4) + 2) + 2 = \dots = 3n/2 - 2$$

Compared to Naïve method, in divide and conquer approach, the number of comparisons is less. However, using the asymptotic notation both of the approaches are represented by $O(n)$.

Algorithm to check whether the given number is Prime or not

Aim:

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

Algorithm:

```

Step 1: Start
Step 2: Read number n
Step 3: Set f=0
Step 4: For i=2 to n-1
Step 5: If n mod i==0 then
Step 6: Set f=1 and break
Step 7: Loop
Step 8: If f=0 then
print 'The given number is prime'
else
print 'The given number is not prime'

```

Step 9: Stop

Program code

```
#include<stdio.h>
#include<conio.h>
void main( )
{
clrscr();
int n,i,f=0;
printf("Enter the number: ");
scanf("%d",&n);
for(i=2;i<n;i++)
{
if(n%i==0)
{
f=1;
break;
}
}
if(f==0)
printf("The given number is prime");
else
printf("The given number is not prime");
getch();
}
```

Output

```
Enter the number : 5
The given number is prime
```

Linear Search Algorithm (Sequential Search)

What is Search?

Searching is a process of finding a value in a list of values. In other words, searching is the process of locating the given value position in a list of values.

Searching is the process of finding the given element in a list of elements.

Linear search algorithm finds the given element in a list of elements with **$O(n)$** time complexity where **n** is the total number of elements in the list. This search process starts comparing the search element with the first element in the list. If both are matched then the result is "**element found**" otherwise the search element is compared with the next element in the list. If both are matched, then the result is "**element found**". Otherwise, repeat the same with the next element in the list until the search element is compared with the last element in the list. If that last element also doesn't match with the search element, then the result we get is "**Element not found in the list**". That means, the search element is compared with all the elements in the list sequentially until the match found.

Linear search is implemented using following steps...

- **Step 1:** Read the search element from the user
- **Step 2:** Compare, the search element with the first element in the list.
- **Step 3:** If both are matched, then display "Given element found!!!" and terminate the function
- **Step 4:** If both are not matched, then compare search element with the next element in the list.

- **Step 5:** Repeat steps 3 and 4 until the search element is compared with the last element in the list.
- **Step 6:** If the last element in the list is also not matched, then display "Element not found!!!" and terminate the function.

Example

Consider the following list of element and search element...

list

0	1	2	3	4	5	6	7
65	20	10	55	32	12	50	99

search element **12**

Step 1:

search element (12) is compared with first element (65)

list

0	1	2	3	4	5	6	7
65	20	10	55	32	12	50	99

12

Both are not matching. So move to next element

Step 2:

search element (12) is compared with next element (20)

list

0	1	2	3	4	5	6	7
65	20	10	55	32	12	50	99

12

Both are not matching. So move to next element

Step 3:

search element (12) is compared with next element (10)

list

0	1	2	3	4	5	6	7
65	20	10	55	32	12	50	99

12

Both are not matching. So move to next element

Step 4:

search element (12) is compared with next element (55)

list

0	1	2	3	4	5	6	7
65	20	10	55	32	12	50	99

12

Both are not matching. So move to next element

Step 5:

search element (12) is compared with next element (32)

list

0	1	2	3	4	5	6	7
65	20	10	55	32	12	50	99

12

Both are not matching. So move to next element

Step 6:

search element (12) is compared with next element (12)

list

0	1	2	3	4	5	6	7
65	20	10	55	32	12	50	99

12

Both are matching. So we stop comparing and display element found at index 5.

Program to implement Linear Search Algorithm using C.

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

void main(){
int list[20],size,i,sElement;

printf("Enter size of the list: ");
scanf("%d",&size);

printf("Enter any %d integer values: ",size);
for(i = 0; i < size; i++)
scanf("%d",&list[i]);

printf("Enter the element to be Search: ");
scanf("%d",&sElement);

    // Linear Search Logic
for(i = 0; i < size; i++)
{
if(sElement == list[i])
{
printf("Element is found at %d index", i);
break;
}
}
if(i == size)
printf("Given element is not found in the list!!!");
getch();

}
```

BINARY SEARCH ALGORITHM

Searching is a process of finding a value in a list of values. In other words, searching is the process of locating the given value position in a list of values.

Searching is the process of finding the given element in a list of elements

Binary search algorithm finds the given element in a list of elements with $O(\log n)$ time complexity where n is the total number of elements in the list. The binary search algorithm can be used only with sorted list of elements. That means, binary search can be used only with list of elements that are already arranged in order. The binary search cannot be used with

unordered list of elements. This search process starts comparing the search element with the middle element in the list. If both are matched, then the result is "**element found**". Otherwise, we check whether the search element is smaller or larger than the middle element in the list. If the search element is smaller, then we repeat the same process for left sublist of the middle element. If the search element is larger, then we repeat the same process for right sublist of the middle element. We repeat this process until we find the search element in the list or until we are left with only one element in the sublist. And if that element also doesn't match with the search element, then we get the result as "Element not found in the list".

Binary search is implemented using following steps...

- **Step 1:** Read the search element from the user
- **Step 2:** Find the middle element in the sorted list
- **Step 3:** Compare, the search element with the middle element in the sorted list.
- **Step 4:** If both are matched, then display "Given element found!!!" and terminate the function
- **Step 5:** If both are not matched, then check whether the search element is smaller or larger than middle element.
- **Step 6:** If the search element is smaller than middle element, then repeat steps 2, 3, 4 and 5 for the left sublist of the middle element.
- **Step 7:** If the search element is larger than middle element, then repeat steps 2, 3, 4 and 5 for the right sublist of the middle element.
- **Step 8:** Repeat the same process until we find the search element in the list or until the sublist contains only one element.
- **Step 9:** If that element also doesn't match with the search element, then display "Element not found in the list!!!" and terminate the function.

Example

Consider the following list of elements and search element...

list

0	1	2	3	4	5	6	7	8
10	12	20	32	50	55	65	80	99

search element 12

Step 1:

search element (12) is compared with middle element (50)

list

0	1	2	3	4	5	6	7	8
10	12	20	32	50	55	65	80	99

12

Both are not matching. And 12 is smaller than 50. So we search only in the left sublist (i.e. 10, 12, 20 & 32).

list

0	1	2	3	4	5	6	7	8
10	12	20	32	50	55	65	80	99

Step 2:

search element (12) is compared with middle element (12)

list

0	1	2	3	4	5	6	7	8
10	12	20	32	50	55	65	80	99

12

Both are matching. So the result is "Element found at index 1"

search element 80

Step 1:

search element (80) is compared with middle element (50)

list

0	1	2	3	4	5	6	7	8
10	12	20	32	50	55	65	80	99

80

Both are not matching. And 80 is larger than 50. So we search only in the right sublist (i.e. 55, 65, 80 & 99).

list

0	1	2	3	4	5	6	7	8
10	12	20	32	50	55	65	80	99

Step 2:

search element (80) is compared with middle element (65)

list

0	1	2	3	4	5	6	7	8
10	12	20	32	50	55	65	80	99

80

Both are not matching. And 80 is larger than 65. So we search only in the right sublist (i.e. 80 & 99).

list

0	1	2	3	4	5	6	7	8
10	12	20	32	50	55	65	80	99

Step 3:

search element (80) is compared with middle element (80)

0	1	2	3	4	5	6	7	8
10	12	20	32	50	55	65	80	99

Program to implement Binary Search Algorithm using C.

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

void main()
{
    int first, last, middle, size, i, sElement, list[100];
    clrscr();

    printf("Enter the size of the list: ");
    scanf("%d",&size);

    printf("Enter %d integer values in Assending order\n", size);

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

    printf("Enter value to be search: ");
    scanf("%d", &sElement);

    first = 0;
    last = size - 1;
    middle = (first+last)/2;

    while (first <= last) {
        if (list[middle] <sElement)
            first = middle + 1;
        else if (list[middle] == sElement) {
            printf("Element found at index %d.\n",middle);
            break;
        }
        else
            last = middle - 1;

        middle = (first + last)/2;
    }
    if (first > last)
        printf("Element Not found in the list.");
    getch();
}
```

Bubble Sort Algorithm

Bubble Sort is a simple algorithm which is used to sort a given set of n elements provided in form of an array with n number of elements. Bubble Sort compares all the element one by one and sort them based on their values.

If the given array has to be sorted in ascending order, then bubble sort will start by comparing the first element of the array with the second element, if the first element is greater than the second element, it will **swap** both the elements, and then move on to compare the second and the third element, and so on.

If we have total n elements, then we need to repeat this process for $n-1$ times.

It is known as **bubble sort**, because with every complete iteration the largest element in the given array, bubbles up towards the last place or the highest index, just like a water bubble rises up to the water surface.

Sorting takes place by stepping through all the elements one-by-one and comparing it with the adjacent element and swapping them if required.

Implementing Bubble Sort Algorithm

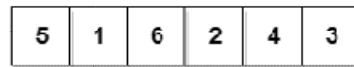
Following are the steps involved in bubble sort(for sorting a given array in ascending order):

1. Starting with the first element(index = 0), compare the current element with the next element of the array.
2. If the current element is greater than the next element of the array, swap them.
3. If the current element is less than the next element, move to the next element. **Repeat Step 1.**

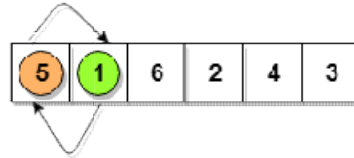
Let's consider an array with values $\{5, 1, 6, 2, 4, 3\}$

Below, we have a pictorial representation of how bubble sort will sort the given array.

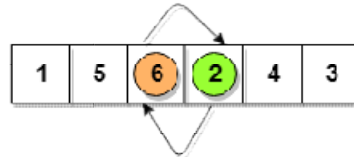
5>1
so interchange



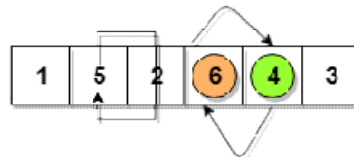
5<6
No swapping



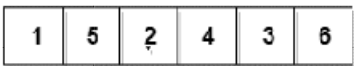
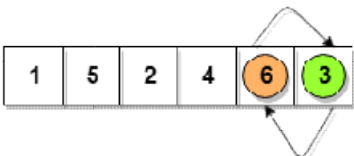
6>2
so interchange



6>4
so interchange



6>3
so interchange



This is first insertion

similarly, after all the
iterations, the array
gets sorted

So as we can see in the representation above, after the first iteration, 6 is placed at the last index, which is the correct position for it.

Similarly after the second iteration, 5 will be at the second last index, and so on.

Time to write the code for bubble sort:

// below we have a simple C program for bubble sort

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

```
void bubbleSort(int arr[], int n)
{
    int i, j, temp;
    for(i = 0; i < n; i++)
    {
        for(j = 0; j < n-i-1; j++)
        {
            if( arr[j] > arr[j+1])
            {
                // swap the elements
                temp = arr[j];
                arr[j] = arr[j+1];
                arr[j+1] = temp;
            }
        }
    }
}
```

```

        arr[j+1] = temp;
    }
}

// print the sorted array
printf("Sorted Array: ");
for(i = 0; i < n; i++)
{
    printf("%d ", arr[i]);
}

}

int main()
{
    int arr[100], i, n, step, temp;
    // ask user for number of elements to be sorted
    printf("Enter the number of elements to be sorted: ");
    scanf("%d", &n);
    // input elements if the array
    for(i = 0; i < n; i++)
    {
        printf("Enter element no. %d: ", i+1);
        scanf("%d", &arr[i]);
    }
    // call the function bubbleSort
    bubbleSort(arr, n);

    return 0;
}

```

Although the above logic will sort an unsorted array, still the above algorithm is not efficient because as per the above logic, the outer **for** loop will keep on executing for **6** iterations even if the array gets sorted after the second iteration. So, we can clearly optimize our algorithm.

Optimized Bubble Sort Algorithm

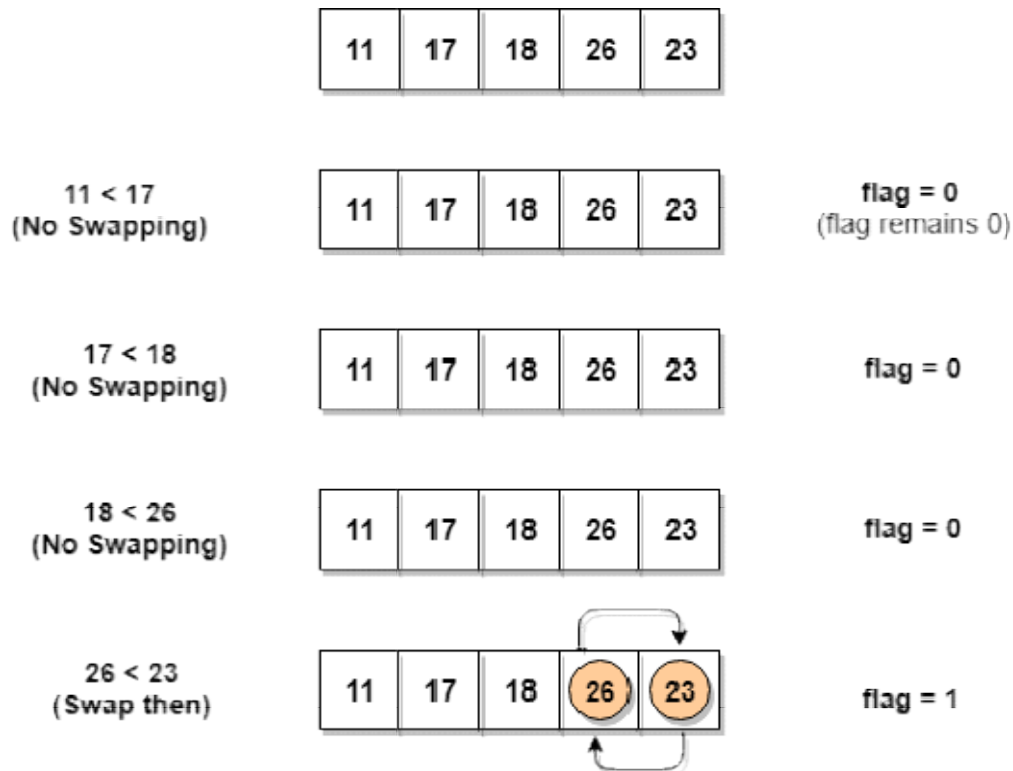
To optimize our bubble sort algorithm, we can introduce a **flag** to monitor whether elements are getting swapped inside the inner **for** loop.

Hence, in the inner **for** loop, we check whether swapping of elements is taking place or not, everytime.

If for a particular iteration, no swapping took place, it means the array has been sorted and we can jump out of the **for** loop, instead of executing all the iterations.

Let's consider an array with values {11, 17, 18, 26, 23}

Below, we have a pictorial representation of how the optimized bubble sort will sort the given array.



As we can see, in the first iteration, swapping took place, hence we updated our **flag** value to **1**, as a result, the execution enters the **for** loop again. But in the second iteration, no swapping will occur, hence the value of **flag** will remain **0**, and execution will break out of loop.

```
// below we have a simple C program for bubble sort
```

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

```
void bubbleSort(int arr[], int n)
{
```

```
    int i, j, temp;
```

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

```
    {
```

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

```
        {
```

```
            // introducing a flag to monitor swapping
```

```
            int flag = 0;
```

```
            if( arr[j] > arr[j+1])
```

```
            {
```

```
                // swap the elements
```

```
                temp = arr[j];
```

```
                arr[j] = arr[j+1];
```

```

        arr[j+1] = temp;
        // if swapping happens update flag to 1
        flag = 1;
    }
}
// if value of flag is zero after all the iterations of inner
loop
    // then break out
    if(!flag)
    {
        break;
    }
}

// print the sorted array
printf("Sorted Array: ");
for(i = 0; i < n; i++)
{
    printf("%d  ", arr[i]);
}
}

int main()
{
    int arr[100], i, n, step, temp;
    // ask user for number of elements to be sorted
    printf("Enter the number of elements to be sorted: ");
    scanf("%d", &n);
    // input elements if the array
    for(i = 0; i < n; i++)
    {
        printf("Enter element no. %d: ", i+1);
        scanf("%d", &arr[i]);
    }
    // call the function bubbleSort
    bubbleSort(arr, n);

    return 0;
}

```

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

Complexity Analysis of Bubble Sort

In Bubble Sort, $n-1$ comparisons will be done in the 1st pass, $n-2$ in 2nd pass, $n-3$ in 3rd pass and so on. So the total number of comparisons will be,

$$(n-1) + (n-2) + (n-3) + \dots + 3 + 2 + 1$$

$$\text{Sum} = n(n-1)/2$$

$$\text{i.e. } O(n^2)$$

Hence the **time complexity** of Bubble Sort is $O(n^2)$.

The main advantage of Bubble Sort is the simplicity of the algorithm.

The **space complexity** for Bubble Sort is $O(1)$, because only a single additional memory space is required i.e. for **temp** variable.

Also, the **best case time complexity** will be $O(n)$, it is when the list is already sorted.

Following are the Time and Space complexity for the Bubble Sort algorithm.

- Worst Case Time Complexity [Big-O]: $O(n^2)$
- Best Case Time Complexity [Big-omega]: $O(n)$
- Average Time Complexity [Big-theta]: $O(n^2)$
- Space Complexity: $O(1)$

Insertion Sort Algorithm

Sorting is the process of arranging a list of elements in a particular order (Ascending or Descending).

Insertion sort algorithm arranges a list of elements in a particular order. In insertion sort algorithm, every iteration moves an element from unsorted portion to sorted portion until all the elements are sorted in the list.

Step by Step Process

The insertion sort algorithm is performed using following steps...

- **Step 1** - Assume that first element in the list is in sorted portion and all the remaining elements are in unsorted portion.

- **Step 2:** Take first element from the unsorted portion and insert that element into the sorted portion in the order specified.
- **Step 3:** Repeat the above process until all the elements from the unsorted portion are moved into the sorted portion.

Following is the sample code for insertion sort...

Insertion Sort Logic

```
//Insertion sort logic
for i = 1 to size-1 {
    temp = list[i];
    j = i-1;
    while ((temp < list[j]) && (j > 0)) {
        list[j] = list[j-1];
        j = j - 1;
    }
    list[j] = temp;
}
```

Example

Consider the following unsorted list of elements...

15	20	10	30	50	18	5	45
----	----	----	----	----	----	---	----

Asume that sorted portion of the list empty and all elements in the list are in unsorted portion of the list as shown in the figure below...

Sorted	Unsorted
	15 20 10 30 50 18 5 45

Move the first element 15 from unsorted portion to sorted portion of the list.

Sorted	Unsorted
15	20 10 30 50 18 5 45

To move element 20 from unsorted to sorted portion, Compare 20 with 15 and insert it at correct position

Sorted	Unsorted
15 20	10 30 50 18 5 45

To move element 10 from unsorted to sorted portion, Compare 10 with 20 and it is smaller so swap. Then compare 10 with 15 again smaller swap. And 10 is insert at its correct position in sorted portion of the list.

Sorted	Unsorted
10 15 20	30 50 18 5 45

To move element 30 from unsorted to sorted portion, Compare 30 with 20, 15 and 10. And it is larger than all these so 30 is directly inserted at last position in sorted portion of the list.

Sorted	Unsorted
10 15 20 30	50 18 5 45

To move element 50 from unsorted to sorted portion, Compare 50 with 30, 20, 15 and 10. And it is larger than all these so 50 is directly inserted at last position in sorted portion of the list.

Sorted	Unsorted
10 15 20 30 50	18 5 45

To move element 18 from unsorted to sorted portion, Compare 18 with 30, 20 and 15. Since 18 is larger than 15, move 20, 30 and 50 one position to the right in the list and insert 18 after 15 in the sorted portion.

Sorted	Unsorted
10 15 18 20 30 50	5 45

To move element 5 from unsorted to sorted portion, Compare 5 with 50, 30, 20, 18, 15 and 10. Since 5 is smaller than all these element, move 10, 15, 18, 20, 30 and 50 one position to the right in the list and insert 5 at first position in the sorted list.

Sorted	Unsorted
5 10 15 18 20 30 50	

Complexity of the Insertion Sort Algorithm

To sort an unsorted list with ' n ' number of elements, we need to make $(1+2+3+\dots+n-1) = (n(n-1))/2$ number of comparisons in the worst case. If the list is already sorted then it requires ' n ' number of comparisons.

Worst Case : $O(n^2)$

Best Case : $\Omega(n)$

Average Case : $\Theta(n^2)$

Implementaion of Insertion Sort Algorithm using C Programming Language

```
#include<stdio.h>
#include<conio.h>
void main(){
int size, i, j, temp, list[100];

printf("Enter the size of the list: ");
scanf("%d", &size);

printf("Enter %d integer values: ", size);
for (i = 0; i < size; i++)
scanf("%d", &list[i]);

    //Insertion sort logic
for (i = 1; i < size; i++) {
temp = list[i];
    j = i - 1;
while ((temp < list[j]) && (j >= 0)) {
list[j + 1] = list[j];
    j = j - 1;
}
list[j + 1] = temp;
}

printf("List after Sorting is: ");
for (i = 0; i < size; i++)
printf(" %d", list[i]);
getch();
```

}

Selection Sort Algorithm

Selection Sort algorithm is used to arrange a list of elements in a particular order (Ascending or Descending). In selection sort, the first element in the list is selected and it is compared repeatedly with all the remaining elements in the list. If any element is smaller than the selected element (for Ascending order), then both are swapped so that first position is filled with smallest element in the sorted order. Next we select the element at second position in the list and it is compared with all the remaining elements in the list. If any element is smaller than the selected element, then both are swapped. This procedure is repeated till the entire list is sorted.

Step by Step Process

The selection sort algorithm is performed using following steps...

- **Step 1** - Select the first element of the list (i.e., Element at first position in the list).
- **Step 2:** Compare the selected element with all the other elements in the list.
- **Step 3:** In every comparison, if any element is found smaller than the selected element (for Ascending order), then both are swapped.
- **Step 4:** Repeat the same procedure with element in the next position in the list till the entire list is sorted.

Following is the sample code for selection sort...

Selection Sort Logic

```
//Selection sort logic
for(i=0; i<size; i++){
    for(j=i+1; j<size; j++){
        if(list[i] > list[j])
        {
            temp=list[i];
            list[i]=list[j];
            list[j]=temp;
        }
    }
}
```

Consider the following unsorted list of elements...

15	20	10	30	50	18	5	45
----	----	----	----	----	----	---	----

Iteration #1

Select the first position element in the list, compare it with all other elements in the list and whenever we found a smaller element than the element at first position then swap those two elements.

15	20	10	30	50	18	5	45
----	----	----	----	----	----	---	----

15 > 20
FALSE

15	20	10	30	50	18	5	45
----	----	----	----	----	----	---	----

15 > 10
TRUE
SWAP

10	20	15	30	50	18	5	45
----	----	----	----	----	----	---	----

10 > 30
FALSE

10	20	15	30	50	18	5	45
----	----	----	----	----	----	---	----

10 > 50
FALSE

10	20	15	30	50	18	5	45
----	----	----	----	----	----	---	----

10 > 18
FALSE

10	20	15	30	50	18	5	45
----	----	----	----	----	----	---	----

10 > 5
TRUE
SWAP

5	20	15	30	50	18	10	45
---	----	----	----	----	----	----	----

5 > 45
FALSE

List after 1st iteration

5	20	15	30	50	18	10	45
---	----	----	----	----	----	----	----

Iteration #2

Select the second position element in the list, compare it with all other elements in the list and whenever we found a smaller element than the element at first position then swap those two elements.

List after 2nd iteration

5	10	20	30	50	18	15	45
---	----	----	----	----	----	----	----

Iteration #3

Select the third position element in the list, compare it with all other elements in the list and whenever we found a smaller element than the element at first position then swap those two elements.

List after 3rd iteration

5	10	15	30	50	20	18	45
---	----	----	----	----	----	----	----

Iteration #4

Select the fourth position element in the list, compare it with all other elements in the list and whenever we found a smaller element than the element at first position then swap those two elements.

Complexity of the Selection Sort Algorithm

To sort an unsorted list with '**n**' number of elements, we need to make $((n-1)+(n-2)+(n-3)+.....+1) = (n(n-1))/2$ number of comparisons in the worst case. If the list is already sorted then it requires '**n**' number of comparisons.

Worst Case : $O(n^2)$

Best Case : $\Omega(n^2)$

Average Case : $\Theta(n^2)$

Implementaion of Selection Sort Algorithm using C Programming Language

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

void main(){

    intsize,i,j,temp,list[100];
    clrscr();

    printf("Enter the size of the List: ");
    scanf("%d",&size);

    printf("Enter %d integer values: ",size);
    for(i=0; i<size; i++)
        scanf("%d",&list[i]);

    //Selection sort logic

    for(i=0; i<size; i++){
        for(j=i+1; j<size; j++){
            if(list[i] > list[j])
            {
                temp=list[i];
                list[i]=list[j];
                list[j]=temp;
            }
        }
    }
}
```

```
printf("List after sorting is: ");  
for(i=0; i<size; i++)  
printf(" %d",list[i]);  
  
getch();  
}
```

Performance Analysis

What is Performance Analysis of an algorithm?

If we want to go from city "A" to city "B", there can be many ways of doing this. We can go by flight, by bus, by train and also by bicycle. Depending on the availability and convenience, we choose the one which suits us. Similarly, in computer science there are multiple algorithms to solve a problem. When we have more than one algorithm to solve a problem, we need to select the best one. Performance analysis helps us to select the best algorithm from multiple algorithms to solve a problem. When there are multiple alternative algorithms to solve a problem, we analyse them and pick the one which is best suitable for our requirements. Formal definition is as follows...

Performance of an algorithm is a process of making evaluative judgement about algorithms.

It can also be defined as follows...

Performance of an algorithm means predicting the resources which are required to an algorithm to perform its task.

That means when we have multiple algorithms to solve a problem, we need to select a suitable algorithm to solve that problem. We compare algorithms with each other which are solving same problem, to select the

best algorithm. To compare algorithms, we use a set of parameters or set of elements like memory required by that algorithm, execution speed of that algorithm, easy to understand, easy to implement, etc.,

Generally, the performance of an algorithm depends on the following elements...

1. Whether that algorithm is providing the exact solution for the problem?
2. Whether it is easy to understand?
3. Whether it is easy to implement?
4. How much space (memory) it requires to solve the problem?
5. How much time it takes to solve the problem? Etc.,

When we want to analyse an algorithm, we consider only the space and time required by that particular algorithm and we ignore all the remaining elements. Based on this information, performance analysis of an algorithm can also be defined as follows...

Performance analysis of an algorithm is the process of calculating space and time required by that algorithm.

Performance analysis of an algorithm is performed by using the following measures...

1. Space required to complete the task of that algorithm (**Space Complexity**). It includes program space and data space
2. Time required to complete the task of that algorithm (**Time Complexity**)

Space Complexity

What is Space complexity?

When we design an algorithm to solve a problem, it needs some computer memory to complete its execution. For any algorithm, memory is required for the following purposes...

1. To store program instructions.
2. To store constant values.
3. To store variable values.
4. And for few other things like function calls, jumping statements etc,.

Space complexity of an algorithm can be defined as follows...

Total amount of computer memory required by an algorithm to complete its execution is called as space complexity of that algorithm.

Generally, when a program is under execution it uses the computer memory for THREE reasons. They are as follows...

1. **Instruction Space:** It is the amount of memory used to store compiled version of instructions.
2. **Environmental Stack:** It is the amount of memory used to store information of partially executed functions at the time of function call.
3. **Data Space:** It is the amount of memory used to store all the variables and constants.

To calculate the space complexity, we must know the memory required to store different datatype values (according to the compiler). For example, the C Programming Language compiler requires the following...

1. 2 bytes to store Integer value.
2. 4 bytes to store Floating Point value.
3. 1 byte to store Character value.
4. 6 (OR) 8 bytes to store double value.

Consider the following piece of code...

```
int square(int a)
{
    return a*a;
```

```
}
```

In the above piece of code, it requires 2 bytes of memory to store variable 'a' and another 2 bytes of memory is used for **return value**.

That means, totally it requires 4 bytes of memory to complete its execution. And this 4 bytes of memory is fixed for any input value of 'a'. This space complexity is said to be *Constant Space Complexity*.

If any algorithm requires a fixed amount of space for all input values then that space complexity is said to be Constant Space Complexity.

Consider the following piece of code...

Example 2

```
int sum(int A[ ], int n)
{
    int sum = 0, i;
    for(i = 0; i < n; i++)
        sum = sum + A[i];
    return sum;
}
```

In the above piece of code it requires
'n*2' bytes of memory to store array variable 'a[]'
2 bytes of memory for integer parameter 'n'
4 bytes of memory for local integer variables 'sum' and 'i' (2 bytes each)
2 bytes of memory for **return value**.

That means, totally it requires '2n+8' bytes of memory to complete its execution. Here, the total amount of memory required depends on the value of 'n'. As 'n' value increases the space required also increases proportionately. This type of space complexity is said to be *Linear Space Complexity*.

If the amount of space required by an algorithm is increased with the increase of input value, then that space complexity is said to be Linear Space Complexity.

Time Complexity

What is Time complexity?

Every algorithm requires some amount of computer time to execute its instruction to perform the task. This computer time required is called time complexity. Time complexity of an algorithm can be defined as follows...

The time complexity of an algorithm is the total amount of time required by an algorithm to complete its execution.

Generally, running time of an algorithm depends upon the following...

1. Whether it is running on **Single** processor machine or **Multi** processor machine.
2. Whether it is a **32 bit** machine or **64 bit** machine.
3. **Read** and **Write** speed of the machine.
4. The amount of time required by an algorithm to perform **Arithmetic** operations, **logical** operations, **return** value and **assignment** operations etc.,
5. **Input** data

Calculating Time Complexity of an algorithm based on the system configuration is a very difficult task because, the configuration changes from one system to another system. To solve this problem, we must assume a model machine with specific configuration. So that, we can able to calculate generalized time complexity according to that model machine.

To calculate time complexity of an algorithm, we need to define a model machine. Let us assume a machine with following configuration...

1. It is a Single processor machine
2. It is a 32 bit Operating System machine
3. It performs sequential execution
4. It requires 1 unit of time for Arithmetic and Logical operations
5. It requires 1 unit of time for Assignment and Return value
6. It requires 1 unit of time for Read and Write operations

Now, we calculate the time complexity of following example code by using the above defined model machine...

Consider the following piece of code...

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

In above sample code, it requires 1 unit of time to calculate $a+b$ and 1 unit of time to return the value. That means, totally it takes 2 units of time to complete its execution. And it does not change based on the input values of a and b . That means for all input values, it requires same amount of time i.e. 2 units.

If any program requires fixed amount of time for all input values then its time complexity is said to be Constant Time Complexity.

Consider the following piece of code...

```
int sum(int A[], int n)
{
    int sum = 0, i;
    for(i = 0; i < n; i++)
        sum = sum + A[i];
    return sum;
}
```

For the above code, time complexity can be calculated as follows...

int sumOfList(int A[], int n)	Cost Time require for line (Units)	Repeataction No. of Times Executed	Total Total Time required in worst case
{			
int sum = 0, i;	1	1	1
for(i = 0; i < n; i++)	$1 + 1 + 1$	$1 + (n+1) + n$	$2n + 2$
sum = sum + A[i];	2	n	2n
return sum;	1	1	1
}			
4n + 4 Total Time required			

In above calculation **Cost** is the amount of computer time required for a single operation in each line. **Repeataction** is the amount of computer time required by each operation for all its

repetations.

Total is the amount of computer time required by each operation to execute. So above code requires ' **$4n+4$** ' **Units** of computer time to complete the task. Here the exact time is not fixed. And it changes based on the **n** value. If we increase the **n** value then the time required also increases linearly.

Totally it takes ' **$4n+4$** ' units of time to complete its execution and it is **Linear Time Complexity**.

If the amount of time required by an algorithm is increased with the increase of input value then that time complexity is said to be **Linear Time Complexity**.

Asymptotic Notations

What is Asymptotic Notation?

Whenever we want to perform analysis of an algorithm, we need to calculate the complexity of that algorithm. But when we calculate complexity of an algorithm it does not provide exact amount of resource required. So instead of taking exact amount of resource we represent that complexity in a general form (Notation) which produces the basic nature of that algorithm. We use that general form (Notation) for analysis process.

Asymptotic notation of an algorithm is a mathematical representation of its complexity.

Note - In asymptotic notation, when we want to represent the complexity of an algorithm, we use only the most significant terms in the complexity of that algorithm and ignore least significant terms in the complexity of that algorithm (Here complexity can be Space Complexity or Time Complexity).

For example, consider the following time complexities of two algorithms...

- **Algorithm 1 : $5n^2 + 2n + 1$**
- **Algorithm 2 : $10n^2 + 8n + 3$**

Generally, when we analyze an algorithm, we consider the time complexity for larger values of input data (i.e. ' n ' value). In above two time complexities, for larger value of ' n ' the term ' $2n + 1$ ' in algorithm 1 has least significance than the term ' $5n^2$ ', and the term ' $8n + 3$ ' in algorithm 2 has least significance than the term ' $10n^2$ '. Here, for larger value of ' n ' the value of most significant terms ($5n^2$ and $10n^2$) is very larger than the value of least significant terms ($2n + 1$ and $8n + 3$). So for larger value of ' n ' we ignore the least significant terms to represent overall time required by an algorithm. In asymptotic notation, we use only the most significant terms to represent the time complexity of an algorithm.

Majorly, we use THREE types of Asymptotic Notations and those are as follows...

1. Big - Oh (O)
2. Big - Omega (Ω)
3. Big - Theta (Θ)

Big - Oh Notation (O)

Big - Oh notation is used to define the **upper bound** of an algorithm in terms of Time Complexity.

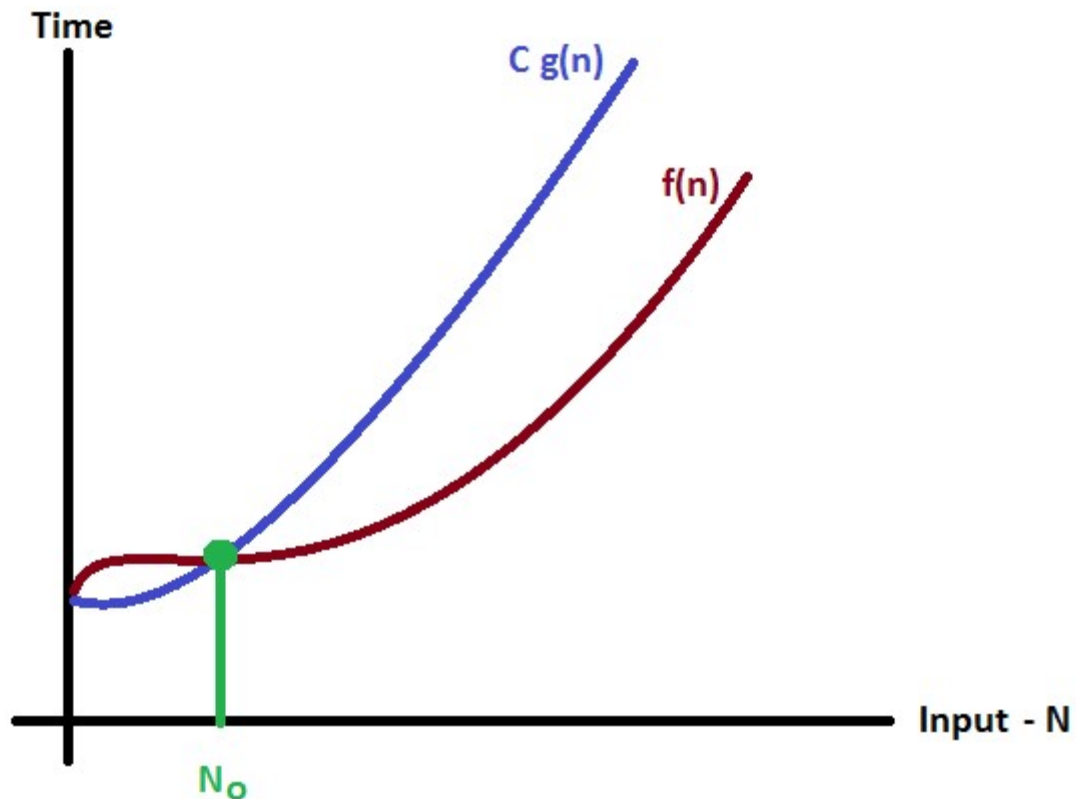
That means Big - Oh notation always indicates the maximum time required by an algorithm for all input values. That means Big - Oh notation describes the worst case of an algorithm time complexity.

Big - Oh Notation can be defined as follows...

Consider function $f(n)$ as time complexity of an algorithm and $g(n)$ is the most significant term. If $f(n) \leq C g(n)$ for all $n \geq n_0$, $C > 0$ and $n_0 \geq 1$. Then we can represent $f(n)$ as $O(g(n))$.

$$f(n) = O(g(n))$$

Consider the following graph drawn for the values of $f(n)$ and $C g(n)$ for input (n) value on X-Axis and time required is on Y-Axis



In above graph after a particular input value n_0 , always $C g(n)$ is greater than $f(n)$ which indicates the algorithm's upper bound.

Example

Consider the following $f(n)$ and $g(n)$...

$$\begin{array}{lclclcl} f(n) & = & 3n & + & 2 \\ g(n) & = & n & & \end{array}$$

If we want to represent $f(n)$ as $O(g(n))$ then it must satisfy $f(n) \leq C g(n)$ for all values of $C > 0$ and $n_0 \geq 1$

$$f(n) \leq C g(n)$$

$$\Rightarrow 3n + 2 \leq C n$$

Above condition is always TRUE for all values of $C = 4$ and $n \geq 2$.

By using Big - Oh notation we can represent the time complexity as follows...

$$3n + 2 = O(n)$$

Big - Omega Notation (Ω)

Big - Omega notation is used to define the **lower bound** of an algorithm in terms of Time Complexity.

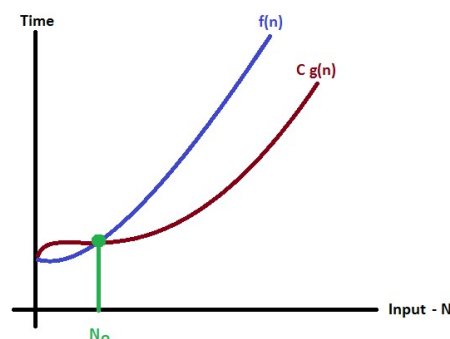
That means Big - Omega notation always indicates the minimum time required by an algorithm for all input values. That means Big - Omega notation describes the best case of an algorithm time complexity.

Big - Omega Notation can be defined as follows...

Consider function $f(n)$ as time complexity of an algorithm and $g(n)$ is the most significant term. If $f(n) \geq C g(n)$ for all $n \geq n_0$, $C > 0$ and $n_0 \geq 1$. Then we can represent $f(n)$ as $\Omega(g(n))$.

$$f(n) = \Omega(g(n))$$

Consider the following graph drawn for the values of $f(n)$ and $C g(n)$ for input (n) value on X-Axis and time required is on Y-Axis



In above graph after a particular input value n_0 , always $C g(n)$ is less than $f(n)$ which indicates the algorithm's lower bound.

Example

Consider the following $f(n)$ and $g(n)$...

$$f(n) = 3n + 2$$

$$g(n) = n$$

If we want to represent $f(n)$ as $\Omega(g(n))$ then it must satisfy $f(n) \geq C g(n)$ for all values of $C > 0$ and $n_0 \geq 1$

$$f(n) \geq C g(n)$$

$$\Rightarrow 3n + 2 \geq C n$$

Above condition is always TRUE for all values of $C = 1$ and $n \geq 1$.

By using Big - Omega notation we can represent the time complexity as follows...

$$3n + 2 = \Omega(n)$$

Big - Theta Notation (Θ)

Big - Theta notation is used to define the **average bound** of an algorithm in terms of Time Complexity.

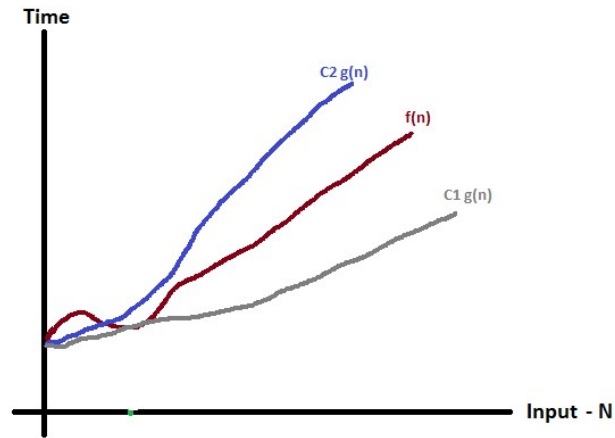
That means Big - Theta notation always indicates the average time required by an algorithm for all input values. That means Big - Theta notation describes the average case of an algorithm time complexity.

Big - Theta Notation can be defined as follows...

Consider function $f(n)$ as time complexity of an algorithm and $g(n)$ is the most significant term. If $C_1 g(n) \leq f(n) \leq C_2 g(n)$ for all $n \geq n_0$, $C_1 > 0$, $C_2 > 0$ and $n_0 \geq 1$. Then we can represent $f(n)$ as $\Theta(g(n))$.

$$f(n) = \Theta(g(n))$$

Consider the following graph drawn for the values of $f(n)$ and $C_1 g(n)$ for input (n) value on X-Axis and time required is on Y-Axis



In above graph after a particular input value n_0 , always $C_1 g(n)$ is less than $f(n)$ and $C_2 g(n)$ is greater than $f(n)$ which indicates the algorithm's average bound.

Example

Consider the following $f(n)$ and $g(n)$...

$$f(n) = 3n + 2$$

$$g(n) = n$$

If we want to represent $f(n)$ as $\Theta(g(n))$ then it must satisfy $C_1 g(n) \leq f(n) \leq C_2 g(n)$ for

all values of $C_1 > 0$, $C_2 > 0$ and $n_0 \geq 1$

$$C_1 g(n) \leq f(n) \leq C_2 g(n)$$

$$\Rightarrow C_1 n \leq 3n + 2 \leq C_2 n$$

Above condition is always TRUE for all values of $C_1 = 1$, $C_2 = 4$ and $n \geq 2$.

By using Big - Theta notation we can represent the time complexity as follows...

$$3n + 2 = \Theta(n)$$

Asymptotic Notations

What is Asymptotic Notation?

Whenever we want to perform analysis of an algorithm, we need to calculate the complexity of that algorithm. But when we calculate complexity of an algorithm it does not provide exact amount of resource required. So instead of taking exact amount of resource we represent that complexity in a general form (Notation) which produces the basic nature of that algorithm. We use that general form (Notation) for analysis process.

Asymptotic notation of an algorithm is a mathematical representation of its complexity.

Note - In asymptotic notation, when we want to represent the complexity of an algorithm, we use only the most significant terms in the complexity of that algorithm and ignore least significant terms in the complexity of that algorithm (Here complexity can be Space Complexity or Time Complexity).

For example, consider the following time complexities of two algorithms...

- **Algorithm 1 : $5n^2 + 2n + 1$**
- **Algorithm 2 : $10n^2 + 8n + 3$**

Generally, when we analyze an algorithm, we consider the time complexity for larger values of input data (i.e. ' n ' value). In above two time complexities, for larger value of ' n ' the term ' $2n + 1$ ' in algorithm 1 has least significance than the term ' $5n^2$ ', and the term ' $8n + 3$ ' in algorithm 2 has least significance than the term ' $10n^2$ '. Here, for larger value of ' n ' the value of most significant terms ($5n^2$ and $10n^2$) is very larger than the value of least significant terms ($2n + 1$ and $8n + 3$). So for larger value of ' n ' we ignore the least significant terms to represent overall time required by an algorithm. In asymptotic notation, we use only the most significant terms to represent the time complexity of an algorithm.

Majorly, we use THREE types of Asymptotic Notations and those are as follows...

1. Big - Oh (O)
2. Big - Omega (Ω)
3. Big - Theta (Θ)

Big - Oh Notation (O)

Big - Oh notation is used to define the **upper bound** of an algorithm in terms of Time Complexity.

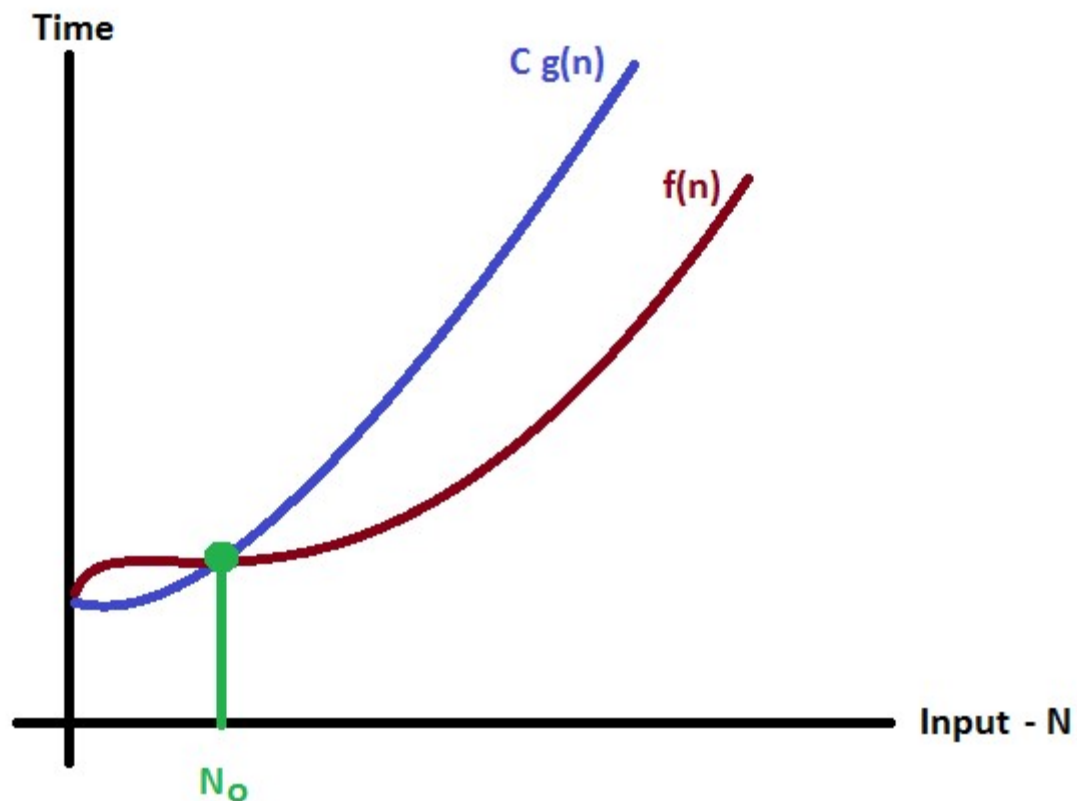
That means Big - Oh notation always indicates the maximum time required by an algorithm for all input values. That means Big - Oh notation describes the worst case of an algorithm time complexity.

Big - Oh Notation can be defined as follows...

Consider function $f(n)$ as time complexity of an algorithm and $g(n)$ is the most significant term. If $f(n) \leq C g(n)$ for all $n \geq n_0$, $C > 0$ and $n_0 \geq 1$. Then we can represent $f(n)$ as $O(g(n))$.

$$f(n) = O(g(n))$$

Consider the following graph drawn for the values of $f(n)$ and $C g(n)$ for input (n) value on X-Axis and time required is on Y-Axis



In above graph after a particular input value n_0 , always $C g(n)$ is greater than $f(n)$ which indicates the algorithm's upper bound.

Example

Consider the following $f(n)$ and $g(n)$...

$$f(n) = 3n + 2$$

$$g(n) = n$$

If we want to represent $f(n)$ as $O(g(n))$ then it must satisfy $f(n) \leq C g(n)$ for all values of $C > 0$ and $n_0 \geq 1$

$$f(n) \leq C g(n)$$

$$\Rightarrow 3n + 2 \leq C n$$

Above condition is always TRUE for all values of $C = 4$ and $n \geq 2$.

By using Big - Oh notation we can represent the time complexity as follows...

$$3n + 2 = O(n)$$

Big - Omega Notation (Ω)

Big - Omega notation is used to define the **lower bound** of an algorithm in terms of Time Complexity.

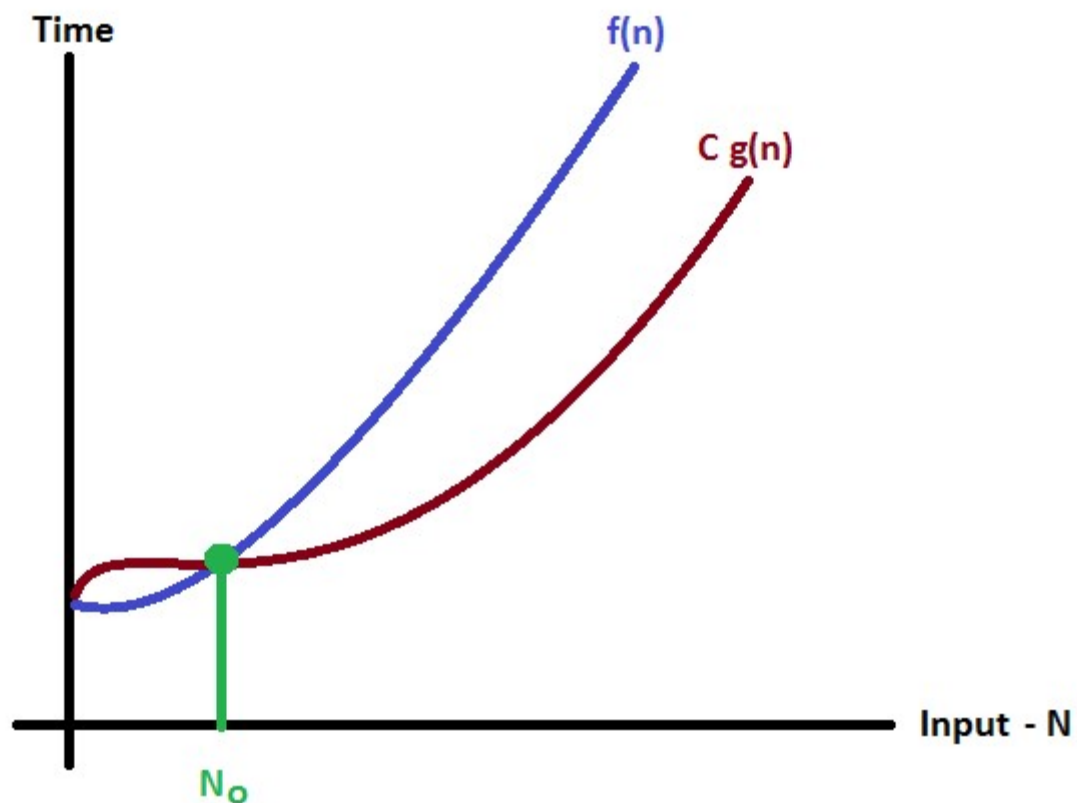
That means Big - Omega notation always indicates the minimum time required by an algorithm for all input values. That means Big - Omega notation describes the best case of an algorithm time complexity.

Big - Omega Notation can be defined as follows...

Consider function $f(n)$ as time complexity of an algorithm and $g(n)$ is the most significant term. If $f(n) \geq C g(n)$ for all $n \geq n_0$, $C > 0$ and $n_0 \geq 1$. Then we can represent $f(n)$ as $\Omega(g(n))$.

$$f(n) = \Omega(g(n))$$

Consider the following graph drawn for the values of $f(n)$ and $C g(n)$ for input (n) value on X-Axis and time required is on Y-Axis



In above graph after a particular input value n_0 , always $C g(n)$ is less than $f(n)$ which indicates the algorithm's lower bound.

Example

Consider the following $f(n)$ and $g(n)$...

$$f(n) = 3n + 2$$

$$g(n) = n$$

If we want to represent $f(n)$ as $\Omega(g(n))$ then it must satisfy $f(n) \geq C g(n)$ for all values of $C > 0$ and $n \geq 1$

$$f(n) \geq C g(n)$$

$$\Rightarrow 3n + 2 \geq C n$$

Above condition is always TRUE for all values of $C = 1$ and $n \geq 1$.

By using Big - Omega notation we can represent the time complexity as follows...

$$3n + 2 = \Omega(n)$$

Big - Theta Notation (Θ)

Big - Theta notation is used to define the **average bound** of an algorithm in terms of Time Complexity.

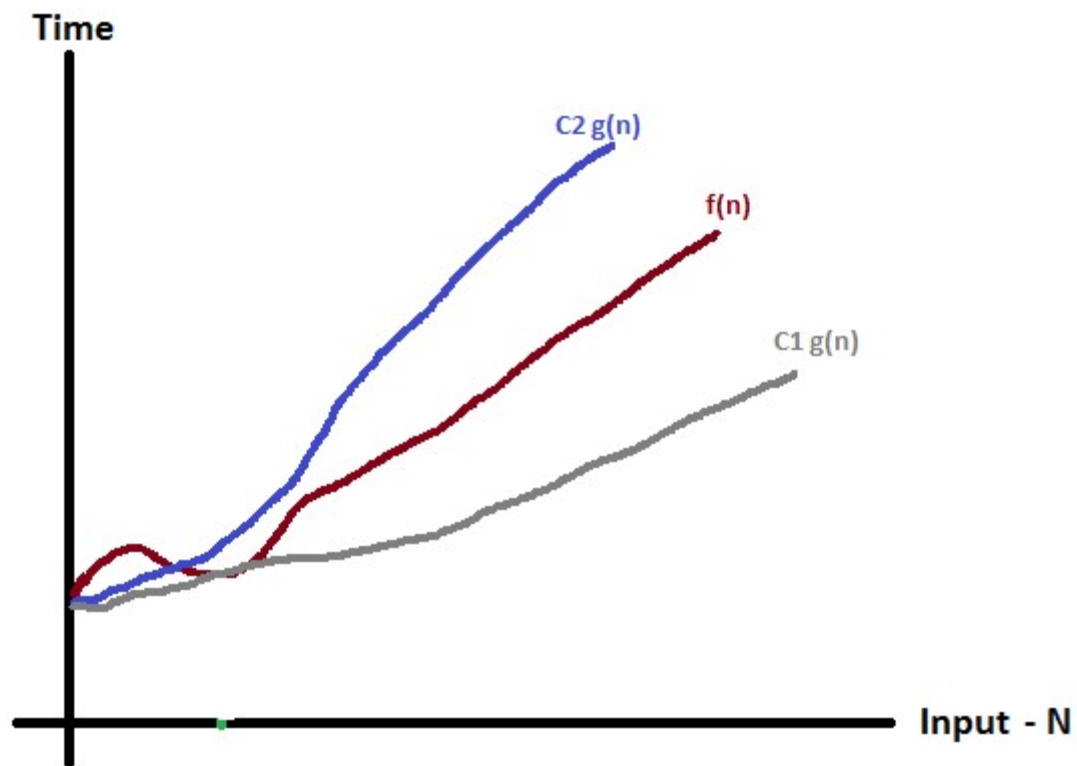
That means Big - Theta notation always indicates the average time required by an algorithm for all input values. That means Big - Theta notation describes the average case of an algorithm time complexity.

Big - Theta Notation can be defined as follows...

Consider function $f(n)$ as time complexity of an algorithm and $g(n)$ is the most significant term. If $C_1 g(n) \leq f(n) \leq C_2 g(n)$ for all $n \geq n_0$, $C_1 > 0$, $C_2 > 0$ and $n_0 \geq 1$. Then we can represent $f(n)$ as $\Theta(g(n))$.

$$f(n) = \Theta(g(n))$$

Consider the following graph drawn for the values of $f(n)$ and $C g(n)$ for input (n) value on X-Axis and time required is on Y-Axis



In above graph after a particular input value n_0 , always $C_1 g(n)$ is less than $f(n)$ and $C_2 g(n)$ is greater than $f(n)$ which indicates the algorithm's average bound.

Example

Consider the following $f(n)$ and $g(n)$...

$$f(n) = 3n + 2$$

$$g(n) = n$$

If we want to represent $f(n)$ as $\Theta(g(n))$ then it must satisfy $C_1 g(n) \leq f(n) \leq C_2 g(n)$ for

all values of $C_1 > 0$, $C_2 > 0$ and $n_0 \geq 1$

$$C_1 g(n) \leq f(n) \leq C_2 g(n)$$

$$\Rightarrow C_1 n \leq 3n + 2 \leq C_2 n$$

Above condition is always TRUE for all values of $C_1 = 1$, $C_2 = 4$ and $n \geq 2$.

By using Big - Theta notation we can represent the time complexity as follows...

$$3n + 2 = \Theta(n)$$

- **Best Case** – Minimum time required for program execution.
- **Average Case** – Average time required for program execution.
- **Worst Case** – Maximum time required for program execution.

Asymptotic Notations

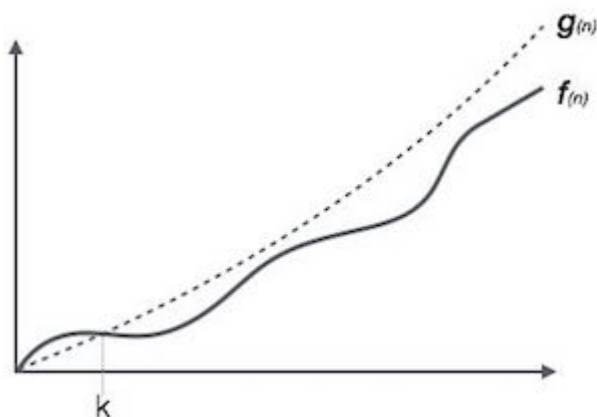
Following are the commonly used asymptotic notations to calculate the running time complexity of an algorithm.

- O Notation
- Ω Notation
- θ Notation

Big Oh Notation, O

The notation $O(n)$ is the formal way to express the upper bound of an algorithm's running time. It measures the worst case time complexity or the

longest amount of time an algorithm can possibly take to complete.

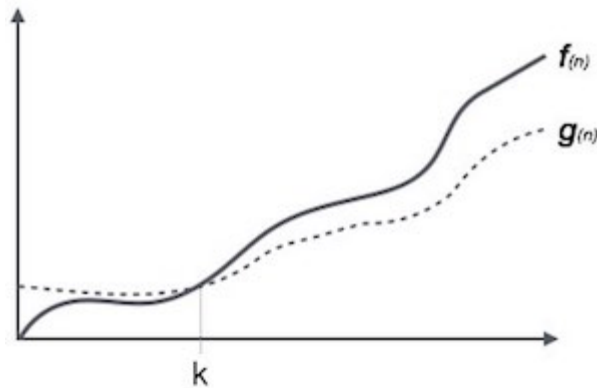


For example, for a function **$f(n)$**

$$O(f(n)) = \{ g(n) : \text{there exists } c > 0 \text{ and } n_0 \text{ such that } f(n) \leq c \cdot g(n) \text{ for all } n > n_0. \}$$

Omega Notation, Ω

The notation $\Omega(n)$ is the formal way to express the lower bound of an algorithm's running time. It measures the best case time complexity or the best amount of time an algorithm can possibly take to complete.



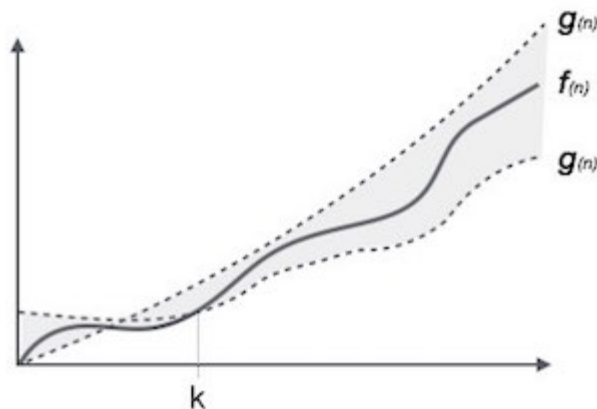
For example, for a function **$f(n)$**

$$\Omega(f(n)) \geq \{ g(n) : \text{there exists } c > 0 \text{ and } n_0 \text{ such that } g(n) \leq c \cdot f(n) \text{ for all } n > n_0. \}$$

Theta Notation, θ

The notation $\theta(n)$ is the formal way to express both the lower bound and the upper bound of an algorithm's running time. It is represented as follows

—



$$\theta(f(n)) = \{ g(n) \text{ if and only if } g(n) = O(f(n)) \text{ and } g(n) = \Omega(f(n)) \text{ for all } n > n_0. \}$$

Common Asymptotic Notations

Following is a list of some common asymptotic notations –

constant	–	$O(1)$
logarithmic	–	$O(\log n)$
linear	–	$O(n)$
$n \log n$	–	$O(n \log n)$
quadratic	–	$O(n^2)$
cubic	–	$O(n^3)$
polynomial	–	$n^{O(1)}$
exponential	–	$2^{O(n)}$