

## UNIT-I

### **Introduction to Programming and Problem Solving**

History of Computers, **Basic organization of a computer:**

ALU, input-output units, memory, program counter,

Introduction to Programming Languages, Basics of a

Computer Program Algorithms, flowcharts (Using Dia

Tool), pseudo code. Introduction to Compilation and

Execution, Primitive Data Types, Variables, and Constants,

Basic Input and Output, Operations, Type Conversion, and

Casting.

**Problem Solving Techniques:** Algorithmic approach,

characteristics of algorithm, **Problem solving strategies:**

Top-down approach, Bottom-up approach, Time and space

complexities of algorithms.

## UNIT-II

### **Control Structures**

Simple sequential programs Conditional Statements (if, if-

else, switch), Loops (for, while, do while) Break and

Continue.

### **UNIT-III**

**Arrays and Strings:** Arrays indexing, memory model, programs with array of integers, two dimensional arrays, Introduction to Strings.

### **UNIT-IV**

#### **Pointers & User Defined Data types**

Pointers, dereferencing and address operators, pointer and address arithmetic, array manipulation using pointers, User-defined data types-Structures and Unions.

### **UNIT-V**

#### **Functions & File Handling**

Introduction to Functions, Function Declaration and Definition, Function call Return Types and Arguments, modifying parameters inside functions using pointers, arrays as parameters. Scope and Lifetime of Variables, Basics of File Handling

## UNIT-I

### **Introduction to Programming and Problem Solving**

History of Computers, Basic organization of a computer: ALU, input-output units, memory, program counter, Introduction to Programming Languages, Basics of a Computer Program Algorithms, flowcharts (Using Dia Tool), pseudo code. Introduction to Compilation and Execution, Primitive Data Types, Variables, and Constants, Basic Input and Output, Operations, Type Conversion, and Casting.

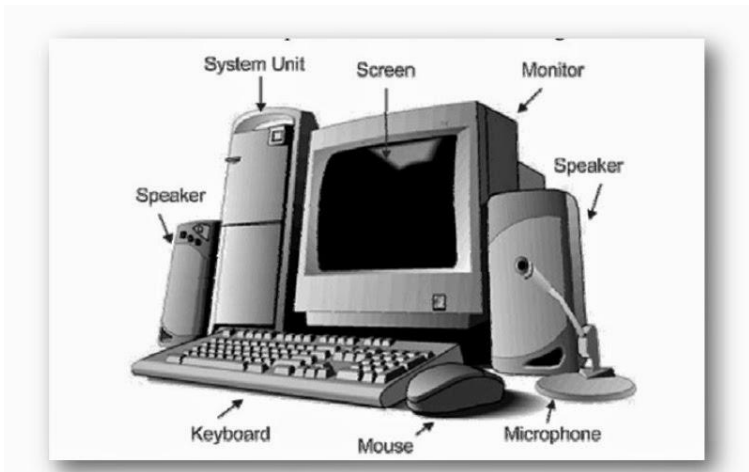
**Problem Solving Techniques:** Algorithmic approach, characteristics of algorithm, Problem solving strategies: Top-down approach, Bottom-up approach, Time and space complexities of algorithms.

### **Introduction**

#### **What is a computer?**

Computer is a high-speed electronic device, which is capable of performing logical and arithmetical operations. It accepts data as input from the user processes and gives desirable output.

In another word “computer is an electronic device which can store and retrieve information and perform mathematical and logical calculation



### **Advantages of Computer**

Computer has become the life-style of mankind because of its special features, which can be categorized as below:

**(a) Speed:** The first feature of computer is speed. It is a very fast device, which can perform any type of task in a fraction of seconds. When performing a particular task for hours together, it can maintain the same speed till the end of it. The speed of a computer can be measured in Pico seconds that is nothing but trillionth part of a second.

Mili second = 100th of sec.

Microsecond = 1 millionth of sec.

Nano Second = 1 billionth of sec.

Pico Second = 1000th of nano sec.

**(b) Accuracy:** The word accuracy means correctness of result. The accuracy with which a computer performs calculation or process data is very high. A computer never gives any wrong information unless and until the user does because a computer does all the operations using electronic circuitry performing millions of operations every second. These circuits can run error free for hours together.



**(c) Storage:** Computer stores many of our records and files in its memory. We can easily retrieve our files & records from the computer.

**(d) Versatility:** A computer is popular today because it can perform different type of works like used for educational use, for research, in banks and railways, in film and animation making.

**(e) Automation:** A computer can perform a particular work continuously for hours together without any human intervention.

**(f) Diligence:** A computer does not suffer from the human traits of tiredness or bored. If 3 million calculations have to be performed, it will perform the 3rd million calculations with exactly the speed and accuracy as the first one.

### **Difference between calculator and computer**

 <b>Computer</b>	 <b>Calculator</b>
<ol style="list-style-type: none"> <li>1. Computer has high Memory capacity.</li> <li>2. We can store data and information in computer.</li> <li>3. It can perform Mathematical and Logical Calculation.</li> <li>4. It calculates up to many digits.</li> <li>5. Its Size is larger than a general Calculator.</li> </ol>	<ol style="list-style-type: none"> <li>1. Its Memory Capacity is very low.</li> <li>2. We can't store any data and information in Calculator.</li> <li>3. Calculator can perform only Mathematical calculations</li> <li>4. It calculates up to limited number of digits.</li> <li>5. Its Size is smaller than a general Computer.</li> </ol>

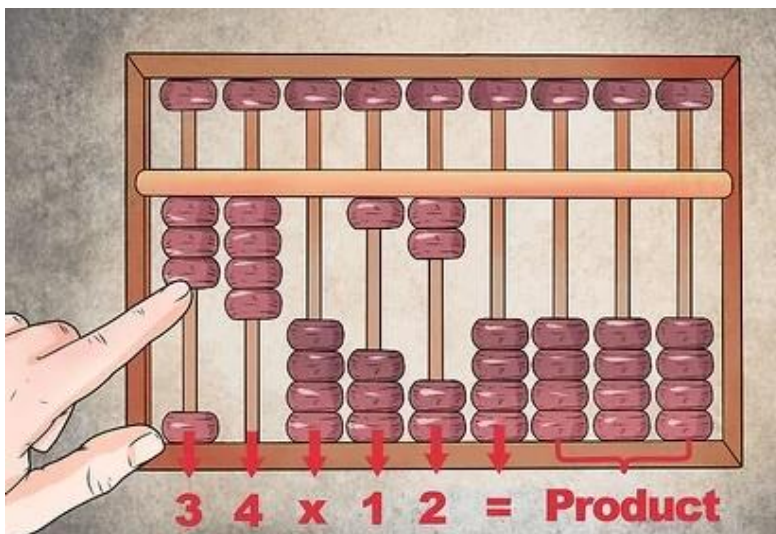
### **1.1 History of Computers**

The first counting device was used by the primitive people. They used sticks, stones and bones as counting tools. As human mind and technology improved with time more computing devices were developed. Some of the popular computing devices starting with the first to recent ones are described below:

#### **a) Abacus**

The history of computer begins with the birth of abacus which is believed to be the first computer. It is said that Chinese invented Abacus around 4,000 years ago.

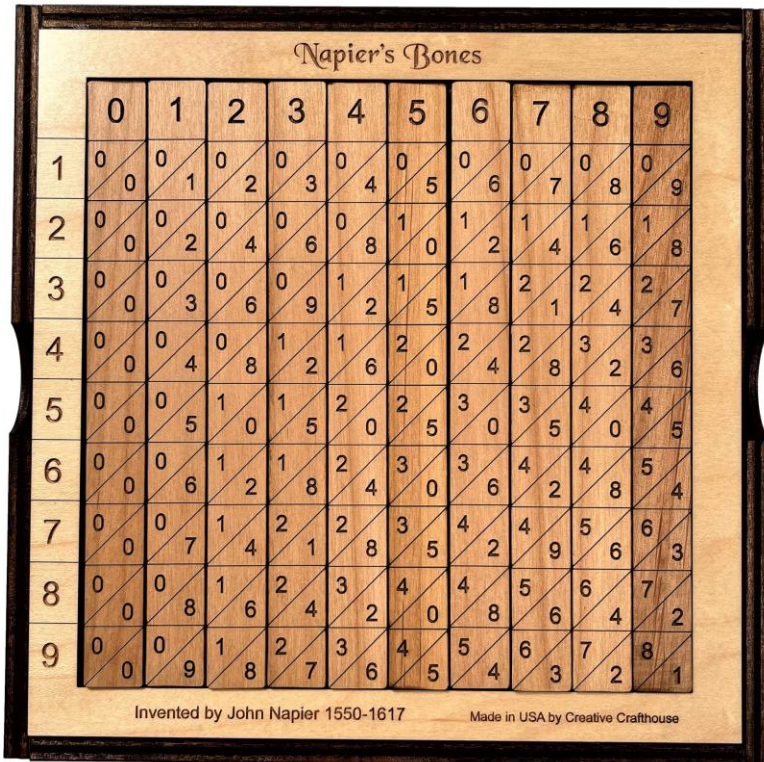
It was a wooden rack which has metal rods with beads mounted on them. The beads were moved by the abacus operator according to some rules to perform arithmetic calculations. Abacus is still used in some countries like China, Russia and Japan. An image of this tool is shown below:



### **b) Napier's Bones**

It was a manually-operated calculating device which was invented by John Napier (1550-1617) of Merchiston. In this calculating tool, he used 9 different ivory strips or bones marked with numbers to multiply and divide. So, the tool

became known as "Napier's Bones. It was also the first machine to use the decimal point.

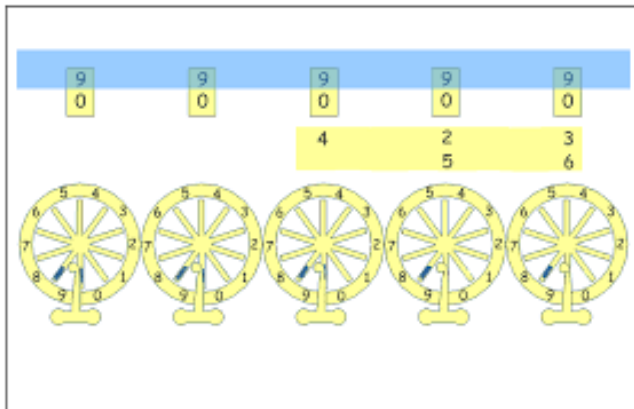


### c) Pascaline

It is also called an Arithmetic Machine or Adding Machine. A French mathematician-philosopher Blaise Pascal invented this between 1642 and 1644. It was the first mechanical and automatic calculator. It is invented by Pascal to help his father, a tax accountant in his work or calculation. It could



perform addition and subtraction in quick time. It was basically a wooden box with a series of gears and wheels. It is worked by rotating wheel like when a wheel is rotated one revolution, it rotates the neighbouring wheel and a series of windows is given on the top of the wheels to read the totals.



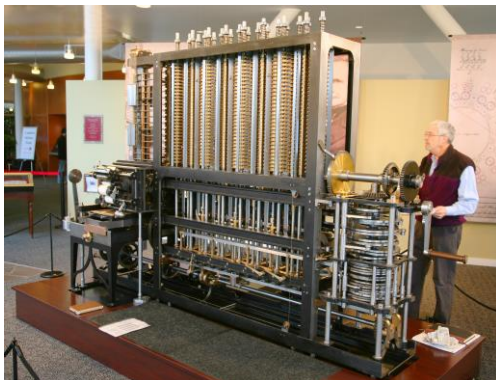
#### **d) Stepped Reckoner or Leibniz wheel**

In 1673, a German mathematician-philosopher named Gottfried Wilhelm Leibniz improved on Pascal's invention to create this apparatus. It was a digital mechanical calculator known as the stepped reckoner because it used fluted drums instead of gears.



### e) Difference Engine

In the early 1820s, Charles Babbage created the Difference Engine. It was a mechanical computer that could do basic computations. It was a steam-powered calculating machine used to solve numerical tables such as logarithmic tables.



### f) Analytical Engine

Charles Babbage created another calculating machine, the Analytical Engine, in 1830. It was a mechanical computer

that took input from punch cards. It was capable of solving any mathematical problem and storing data in an indefinite memory.



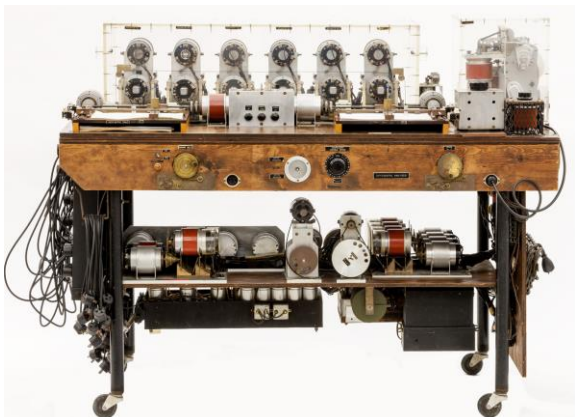
### **g) Tabulating machine**

An American Statistician – Herman Hollerith invented this machine in the year 1890. Tabulating Machine was a punch card-based mechanical tabulator. It could compute statistics and record or sort data or information. Hollerith began manufacturing these machines in his company, which ultimately became International Business Machines (IBM) in 1924.



### **h) Differential Analyzer**

Vannevar Bush introduced the first electrical computer, the Differential Analyzer, in 1930. This machine is made up of vacuum tubes that switch electrical impulses in order to do calculations. It was capable of performing 25 calculations in a matter of minutes.



## **i) Mark I**

Howard Aiken planned to build a machine in 1937 that could conduct massive calculations or calculations using enormous numbers. The Mark I computer was constructed in 1944 as a collaboration between IBM and Harvard.



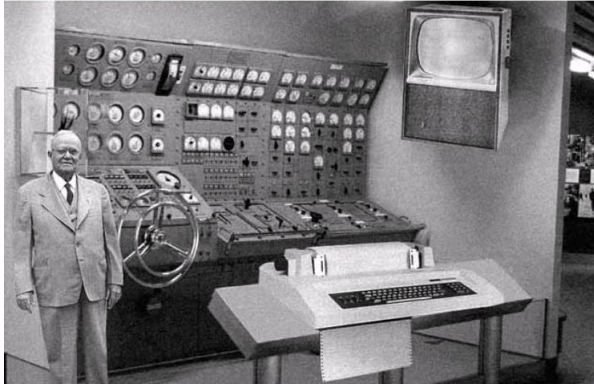
## **1.2 Computer Generations**

The computer has been grouped into five generations.

### **1. First generation**

The first generation computers (1951-1958) used vacuum tubes technology. The first computers are heavy, occupying much space. It consumed large amount of power. It did not have the stored program concepts.

The first computer named was ENIAC (Electronic Numerical Integrator and Calculator).



## **2. Second generation**

The second generation computers (1959-1964) used transistors technology. Transistors helped in developing smaller and more reliable computers. It is used less power and generated less heat than the first generation.



## **3. Third generation**

The third generation computers (1964-1970) use IC(Integrated Circuit) technology. The IC contained many

components fused together on a single chip. These computers are very reliable and could store programs.

The IC's are classified into four types

1. SSI (Small Scale Integrated Circuit)
2. MSI (Medium Scale Integrated Circuit)
3. LSI (Large Scale Integrated Circuit)
4. VLSI (Very Large Scale Integrated Circuit).



#### **4. Fourth generation:**

The fourth generation computers (1971 on words) used Microprocessor technology, hence these computers are called as micro computers or micros. It increases speed, greater reliability, and large storage.



## **5. Fifth generation:**

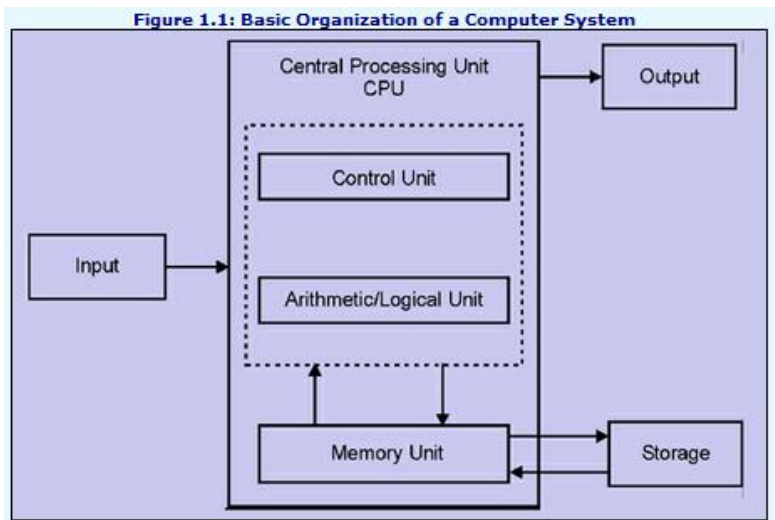
In this generation, AI (Artificial Intelligence) technology is used.





### 1.3 Block diagram of a Computer

The basic organization of a computer system is the processing unit, memory unit, and input-output devices. The processing unit controls all the functions of the computer system. It is the brain of the computer e.g. CPU. The memory unit consists of two units. One is an arithmetic unit and the other is a logic unit. Input devices are those devices through which end-users can send messages to computers e.g. keyboard, mouse, etc. Output devices are those devices through which end-users get output from computers e.g. monitors.



**1. Input Unit:** - Usually a keyboard and mouse is the input. Using these input devices we have to enter data and instructions into a computer. Input unit is provided for man-to-machine communication. It accept data in human readable form, convert it into machine readable form and sends it to CPU.

**2. Central Processing Unit:** - It is the heart of the computer. The component that actually executes instructions. It consists of three units.

- a) Control Unit
- b) Arithmetic Logic Unit (ALU)
- c) Memory Unit (MU)

The main function of CPU is

- Controls the sequence of operation as per the stored instructions.
- Issue commands to all parts of the computer.
- Stores data and instructions.
- Process the data and sends results to output.

**a) Control Unit:-** The control unit controls and co-ordinates all operations of the CPU, input and output devices.

- It gives commands to transfer data from input unit to memory unit, and Arithmetic Logical Unit(ALU).

- It stores the program in the memory and access instructions one by one.
- It transfers the results from ALU to the memory unit and output unit.

**b) Arithmetic Logic Unit (ALU) :-** It carries out all arithmetic operations like addition, subtraction, multiplication and division. It also performs all logical operations. The logical operations consists of >, >=, <, <=, ==

**3) Memory Unit: -** It is used to store programs and data. It is mainly two types they are

- Main Memory or primary memory or Immediate Access Storage (IAS), which is part of the CPU.
- Auxiliary memory or secondary storage, which is external to the CPU.

**4). Output Unit:-** Output Unit is provided for machine- to-man communication. It receives the information from CPU in machine readable form and presents it to the user in a desired form. A computer may have one or more output devices depending upon use.

The Visual Display Unit (VDU) or monitor and printers are most commonly used devices.

## **Program Counter**

A program counter (PC) is a CPU register in the computer processor which has the address of the next instruction to be executed from memory. It is a digital counter needed for faster execution of tasks as well as for tracking the current execution point.

A program counter is also known as an instruction counter, instruction pointer, instruction address register or sequence control register.

## **1.4 Introduction to Programming Languages**

### **Program and Programming:**

**Program:** The set of instructions are called a program. For Example, Programmers create programs by writing code that instructs the computer what to do and execute it on special software designed for it such as turbo C for executing 'C' Programs.

**Programming:-** Programming is the implementation of logic to facilitate the specified computing operations and functionality. Thus, in simple words, we can say that the process of writing a program is called Programming.

## **Types of computer languages**

Basically, computer languages are divided into 3 types.

**Machine language:** Created with binary code [0, 1] and they are very difficult for humans. Example: 11100001

**Low level/assembly language:** Created with English-like shortcuts called MNEMONICS. Example: Add, Sub, Subject, Subtract, Subway, Subscribe, Subscript, subordinate

**High-level language:** Created with simple English.

## **Types of programming language**

### **1. Low-level programming language**

Low-level language is machine-dependent (0s and 1s) programming language. The processor runs low-level programs directly without the need of a compiler or interpreter, so the programs written in low-level language can be run very fast.

Low-level language is further divided into two parts -

#### **i. Machine Language**

Machine language is a type of low-level programming language. It is also called as machine code or object code. Machine language is easier to read because it is normally

displayed in binary or hexadecimal form (base 16) form. It does not require a translator to convert the programs because computers directly understand the machine language programs.

The advantage of machine language is that it helps the programmer to execute the programs faster than the high-level programming language.

## **ii. Assembly Language**

Assembly language (ASM) is also a type of low-level programming language that is designed for specific processors. It represents the set of instructions in a symbolic and human-understandable form. It uses an assembler to convert the assembly language to machine language.

The advantage of assembly language is that it requires less memory and less execution time to execute a program.

## **2. High-level programming language**

High-level programming language (HLL) is designed for developing user-friendly software programs and websites. This programming language requires a compiler or interpreter to translate the program into machine language (execute the program).

The main advantage of a high-level language is that it is easy to read, write, and maintain.

High-level programming language includes Python, Java, JavaScript, PHP, C#, C++, Objective C, Cobol, Perl, Pascal, LISP, FORTRAN, and Swift programming language.

A high-level language is further divided into three parts –

### **i. Procedural Oriented programming language**

Procedural Oriented Programming (POP) language is derived from structured programming and based upon the procedure call concept. It divides a program into small procedures called routines or functions.

Procedural Oriented programming language is used by a software programmer to create a program that can be accomplished by using a programming editor like IDE, Adobe Dreamweaver, or Microsoft Visual Studio.

The advantage of POP language is that it helps programmers to easily track the program flow and code can be reused in different parts of the program.

**Example:** C, FORTRAN, Basic, Pascal, etc.

### **ii. Object-Oriented Programming language**

Object-Oriented Programming (OOP) language is based upon the objects. In this programming language, programs

are divided into small parts called objects. It is used to implement real-world entities like inheritance, polymorphism, abstraction, etc in the program to makes the program reusable, efficient, and easy-to-use.

The main advantage of object-oriented programming is that OOP is faster and easier to execute, maintain, modify, as well as debug.

**Example:** C++, Java, Python, C#, etc.

### **iii. Natural language**

Natural language is a part of human languages such as English, Russian, German, and Japanese. It is used by machines to understand, manipulate, and interpret human's language. It is used by developers to perform tasks such as translation, automatic summarization, Named Entity Recognition (NER), relationship extraction, and topic segmentation.

The main advantage of natural language is that it helps users to ask questions in any subject and directly respond within seconds.



### **3. Middle-level programming language**

Middle-level programming language lies between the low-level programming language and high-level programming language. It is also known as the intermediate programming language and pseudo-language.

A middle-level programming language's advantages are that it supports the features of high-level programming, it is a user-friendly language, and closely related to machine language and human language.

**Example:** C, C++, language

### **1.5 What is Software?**

Software is a collection of the program which uses the resources of the Hardware components. A Program is a set of instructions that are designed for a particular task. The set of programs is called software. Let us understand this with an example i.e. Calculator. For each button, there is some program written inside it. That means a calculator is a collection of programs. And we can also say that a Calculator is a software. So, the software is a collection of programs.

## **Types of Software**

The three major types of software are:

- System Software
- Application Software
- Programming Languages

**System software** programs manage and control the hardware of the computer. The system software has direct access to the hardware. System software maintains the computer to run more efficiently. System software can further be classified into the following types:

- Operating System
- Network Operating System
- System Utilities

**Application software** consists of software applications that are useful to the users. Application software is dependent on System software.

These can be divided into:

- General-purpose software
- Application-specific software

**General-purpose software** can be used for any purpose and for any application. For example, MS Office, OpenOffice,

RDBMS like Oracle, and MySQL, Application servers like JBoss, etc.

**Application-specific software** can only be used for a specific purpose. For example, a Banking application developed for a specific bank.

### **Programming Languages**

Programming languages are used to write code, build, test, and debug software applications. Some examples of high-level programming languages are as follows:

- BASIC
- Pascal
- FORTRAN
- C
- C++
- Java
- Kotlin
- MATLAB

## **1.6 Basics of a Computer Program**

We assume you are well aware of English Language, which is a well-known Human Interface Language. English has a predefined grammar, which needs to be followed to write English statements in a correct way. Likewise, most of the Human Interface Languages (Hindi, English, Spanish, French, etc.) are made of several elements like verbs, nouns, adjectives, adverbs, propositions, and conjunctions, etc.

Similar to Human Interface Languages, Computer Programming Languages are also made of several elements. We will take you through the basics of those elements and make you comfortable to use them in various programming languages. These basic elements include –

- Programming Environment
- Basic Syntax
- Data Types
- Variables
- Keywords
- Basic Operators
- Decision Making
- Loops

- Numbers
- Characters
- Arrays
- Strings
- Functions
- File I/O

### a) Algorithms

An algorithm is the **step-by-step** logical procedure for solving a problem. Each step is called an instruction. An algorithm can be written in English like sentences or in any standard representation. The algorithm written in English like language is called “**pseudo code**”.

Properties of algorithm: According to D.E Knuth a pioneer (found) in the computer discipline an algorithm must have the following properties.

- Input
- Output
- Definiteness
- Finiteness
- Effectiveness

**Input:** Every algorithm takes zero or more inputs . Inputs are the numeric or non-numeric quantities that are given to an algorithm.

**Output:** An algorithm produces one or more outputs. If there are no outputs, the algorithm is considered not to have solved any computational problem.

**Definiteness:** Each instruction in an algorithm should be clear and unambiguous (unclear).

**Finiteness:** The algorithm should terminate after a finite number of steps. It should not enter into an infinite loop.

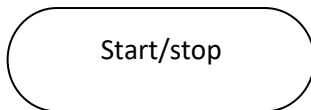
**Effectiveness:** Each step of algorithm must be effective ie; every operation should be done roughly by pen and paper. I.e.; tracing of each step should be possible.

## **b) Flowchart**

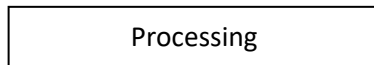
A flowchart is a pictorial representation of an algorithm. It shows the flow of operations in pictorial form and any error in the logic of the problem can be detected very easily. A flow chart uses different shapes of boxes and symbols to denote different types of instructions. These symbols are connected by solid lines with arrow marks to indicate the operation flow.

Normally an algorithm is represented in the form of a flow chart and the flow chart is then expressed in some programming language to prepare a computer program.

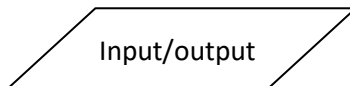
**Flow chart symbols:** A few symbols are needed to indicate the necessary operations in a flow chart. These symbols have been standardized by the American National Standard Institute (ANSI). These symbols are



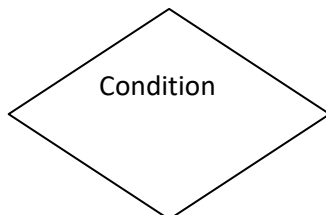
Start and stop commands are written within this symbols.



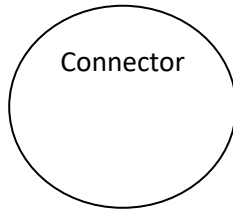
Operations are written with in these symbols.



Read and writing operations are indicated within this symbol.



Control operations are indicated with in this symbol.



One part of the flow chart is connected to another by using this symbol when it does not fit into one sheet.

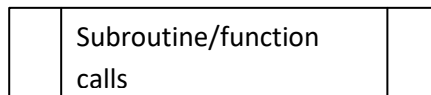


Flow of control is indicated with this symbol.



Group instruction

A group of steps or instructions or a sub-algorithm is indicated with in this symbol.



An activity carried out as part of a function or subroutine is indicated within this symbol.

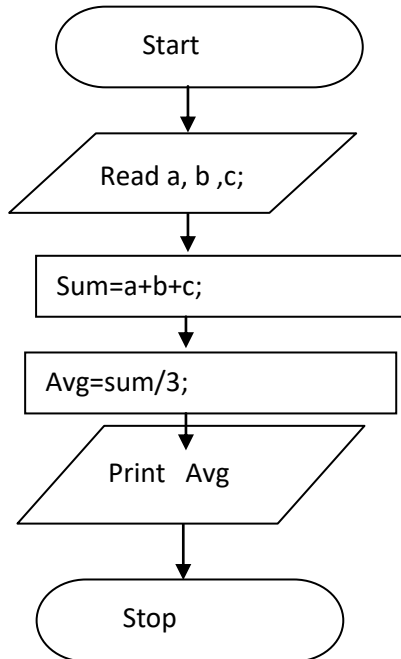


**Example:** Write an algorithm for finding the average of given three numbers and draw the flow chart for the same.

**Sol: Algorithm**

1. Start
2. Read a b and c
3. Compute  $\text{sum} = a + b + c$
4.  $\text{Avg} = \text{sum} / 3$ ;
5. Print avg
6. Stop

**Flow chart:**



## **Advantages of Using Flowcharts**

**Communication:** Flowcharts are a better way of communicating the logic of a system to all concerned or involved.

**Effective analysis:** With the help of flowchart, a problem can be analyzed in a more effective way, therefore reducing cost and wastage of time.

**Proper documentation:** Program flowcharts serve as a good program documentation, which is needed for various purposes, making things more efficient.

**Efficient Coding:** Flowcharts act as a guide or blueprint during the systems analysis and program development phase.

**Proper Debugging:** The flowchart helps in the debugging process.

**Efficient Program Maintenance:** The maintenance of an operating program becomes easy with the help of a flowchart. It helps the programmer to put efforts more efficiently on that part.

## **Disadvantages of Flowchart**

A few disadvantages of Flowcharts are as follows:

- Difficulty in presenting complex programs and tasks.
- No scope for alteration or modification
- Reproduction becomes a problem
- It's a time-consuming process
- Difficult to understand for people who don't know flowchart symbols.
- No man to computer communication.

### **Difficulty in presenting complex programs and tasks:**

Complex tasks cannot be presented through symbols so easily. Sometimes it becomes difficult for even for the expert to present the program having complicated steps. Various logics of the program are difficult to draw in a set or already defined shapes in a flowchart.

**No scope for alteration or modification:** If there is any error found in the process or logic of the flowchart, it is difficult to alter or modify the same. This is because both we have to erase the beginning or end of the program and the whole flowchart is affected.

**Reproduction becomes a problem:** The flowchart symbols cannot be drawn. We have to use different applications like Word or Excel to draw shapes and type words inside those symbols. This recreation becomes difficult as they require shapes to present the whole process.

**It's a time-consuming process:** as we have already seen, reproduction of flowcharts is a problem due to the complexity of shapes and the use of no specific application to give already set up symbols to write the logic of the process or task. Thus it becomes a time-absorbing task.

**Difficult to understand for people who don't know flowchart symbols:** Since not all are experts in understanding the purpose and motive behind using specific symbols in a flowchart it is not understandable by the common people instantly. This proper knowledge and expertise are important to understand the flowchart. Thus communication will become more effective.

**No man to computer communication:** A flowchart is not meant for man to computer communication. Only man can interpret the result of the flowchart.

### **c) Pseudo code**

Pseudo code is structured English for describing algorithms concisely. It is made up of two words, namely, pseudo meaning imitation and code meaning instructions. As the name suggests, pseudo code does not obey the syntax rules of any particular programming language i.e. it is not a real programming code. It allows the designer to focus on main logic without being distracted by programming languages syntax.

#### **Guidelines for Preparing Pseudo code**

- Pseudo code will be enclosed by START (or BEGIN) and STOP (or END).
- Pseudo code should be concise so ignore unnecessary details.
- To accept data from user, generally used statements are INPUT, READ, GET or OBTAIN.
- To display result or any message, generally used statements are PRINT, DISPLAY, or WRITE.
- Generally used keywords are capitalized while preparing pseudo code.

## Example of Pseudo code

### Pseudo code: Calculation of Simple Interest

```
Step 1: START  
Step 2: READ P, T, R  
Step 3: I = P*T*R/100  
Step 4: PRINT I  
Step 5: STOP
```

### Advantages of Pseudo code

- It allows the designer to focus on main logic without being distracted by programming languages syntax.
- Since it is language independent, it can be translated to any computer language code.
- It allows designer to express logic in plain natural language.
- It is easier to write actual code using pseudo code.
- Unlike algorithms, pseudo codes are concise so pseudo codes are more readable and easier to modify.

### Disadvantages of Pseudo code

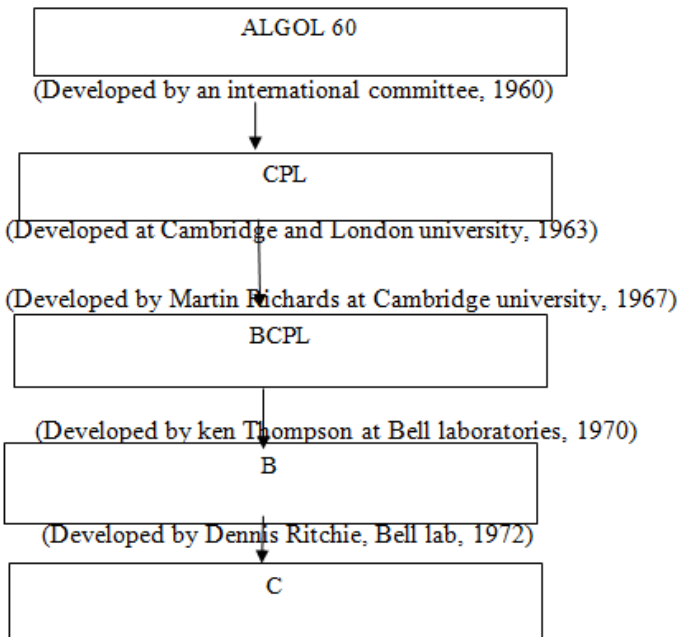
- There are no accepted standards for writing pseudo codes and designer use their own style while writing pseudo codes.

- Pseudo code cannot be compiled and executed so its correctness cannot be verified by using computers.

## 1.7 Fundamentals of C: History

How ‘C’ evolved: The development of C language was a result of the evolution of several languages. This can be called the ‘ancestors’ of C. Those were ALOGOL60, CPL, BCPL, and B.

In 1960 many computer languages, each for a specific purpose, were developed.



1. **ALGOL 60:** ALGOL 60 was a modular and structured language but it did not succeed because **it was found to be too abstract (theoretical) and too general (all-purpose).**
2. **CPL:** The Combined Programming Language was developed at Cambridge University and university of London in 1963. However **it was hard to learn and difficult to implement.**
3. **BCPL:** The basic combined programming language was very close to CPL and developed by Martin Richards at Cambridge University in 1967. **BCPL was too less powerful and too specific and hence it failed.**
4. **B:** The father of C language was the B language developed by ken Thompson of Bell Laboratories in 1970. It was first designed for UNIX. **However, it was machine dependent and a type less language.**
5. **C:** The ‘C’ programming language was invented by Dennis Ritchie in 1972 at Bell Laboratories. **The C was closely linked with UNIX for which it was developed.**



## **Application areas**

1. 'C' is a general purpose programming language and not designed for specific application areas like FORTRAN (Scientific) and COBOL (Business).
2. C is well suited for business as well as scientific applications because it has various features requires for these applications. However it is better suited and widely used for system software like operating systems, compilers, interpreters etc.

## **Why C**

There are several reasons why C is a popular programming language.

- 1. Flexibility:** C is a general purpose language. It can be used for diverse applications. The language itself places no constraints on the programmer.
- 2. Powerful:** It provides a variety of data types, control flow instructions for structured programs and other built in features.

**3. Small size:** C language provides no input/output facilities or file access. These mechanisms are provided by functions. This helps in keeping the language small. C has only 32 keywords which can be described in a small space and learned quickly.

**4. Modular design:** The C code has to be written in functions which can be linked with or called in other programs or applications. C also allows user defined functions to be stored in library files and linked to other programs.

**5. Portability:** A C program written for one computer system can be compiled and run on another with little or no modification.

**6. High level structured language features:** This allows the programmer to concentrate on the logic flow of the code rather than worry about the hardware instructions.

**7. Low level features:** C has a close relationship with the assembly language making it easier to write assembly language code in C program.

**8. Bit engineering:** C provides bit manipulation operators which are a great advantage over other languages.

**9. Use of pointers:** This provides for machine independent address arithmetic.

**10. Efficiency:** A program written in C has development efficiency as well as machine efficient (i.e., faster to execute).

### **Limitations of C language**

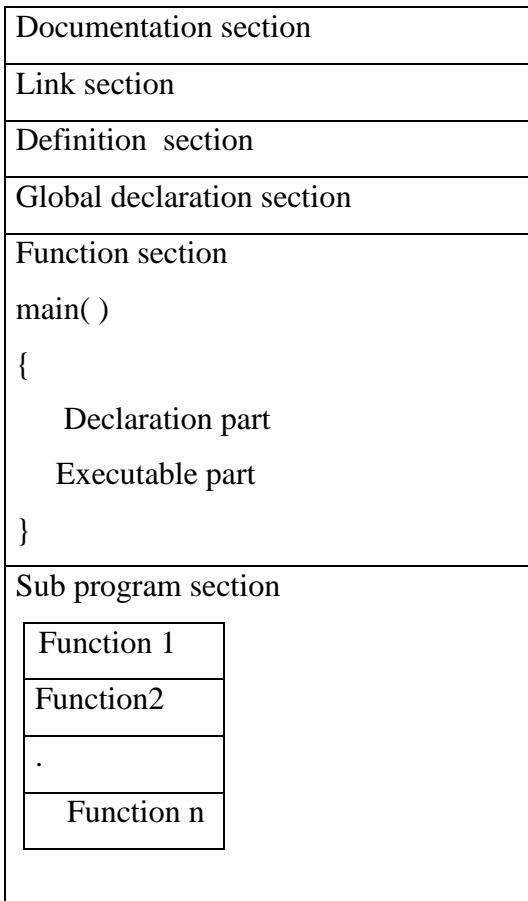
1. It was not suitable for programming of numerical algorithms since it does not provide suitable data structures.
2. C does not perform bound checking on arrays.
3. C is not a strictly type checking language.
4. The order of evaluation of function arguments is not specified by the language.

**Example:-** In the function call `f(i, ++i)` is not defined. Whether the evaluation is left to right or right to left.

5. The order in which operators are evaluated is not specified by the language.

## 1.8 Structure of a C program

A C program consists of functions, one of which is main(). The program begins executing at main(). The basic structure of a C program is as shown:



## **1. Documentation section**

This section consists of comments specifies the name of the program, author and other details. These comments beginning with two characters `/*` and ending with the characters `*/`. No space can be included between these pairs of characters any characters may be included in either upper case or lower case.

## **2. Link section**

This section provides the compiler to link functions from the system library (Ex:- `stdio.h` , `conio.h` , `math.h` ).

## **3. Definition section**

This section defines all symbolic constants.(Ex:- `# define Max 100` or `#define PI 3.14`).

## **4.Global declaration section**

Some variables are used in one or more function, such variables are called global variables and are declared in this section. I.e.; outside of the all the functions.

## **5. Main function section**

Every C program must have at least one function ie; `main( )` function. This main section has two parts.

- Declaration part
- Executable part

The declaration part declares all the variables that are used in the executable part. There is at least one statement in the executable part. These two parts can appear between the opening and closing braces. In all C programs execution begins at this opening and closing braces and ends at this closing brace. All the declaration and executable statements end with a semi colon.

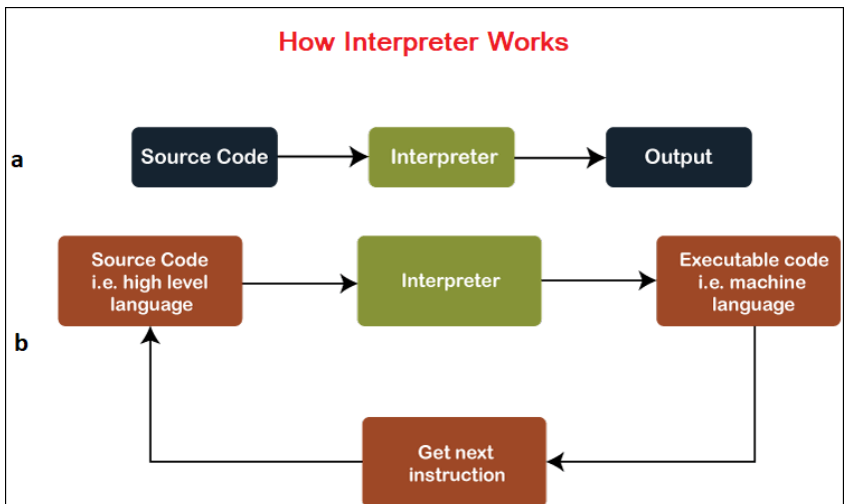
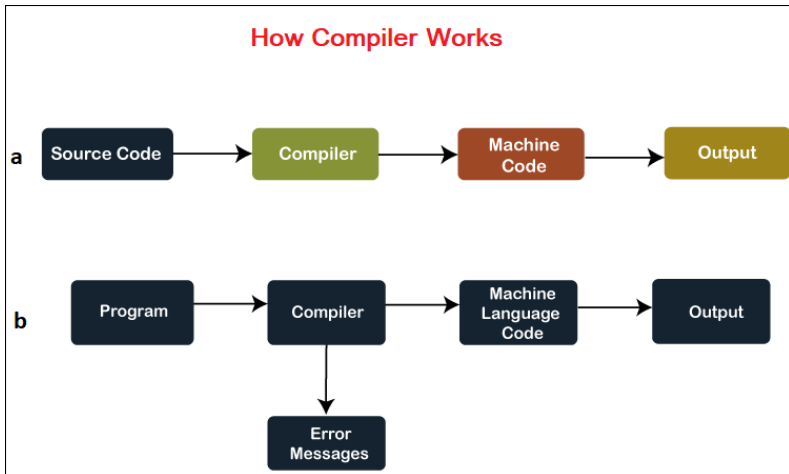
## **6. Subprogram section**

This section contains user defined functions that are mentioned in the main function. User defined functions are generally placed immediately after the main function.

## **1.9 Introduction to Compilation and Execution**

The compilation process in C is converting an understandable human code into a Machine understandable code and checking the syntax and semantics of the code to determine any syntax errors or warnings present in our C program. Suppose we want to execute our C Program written in an IDE (Integrated Development Environment).

In that case, it has to go through several phases of compilation (translation) to become an executable file that a machine can understand.



The compilation and execution process of C can be divided in to multiple steps:

**Preprocessing** - Using a Preprocessor program to convert C source code in expanded source code. "#includes" and "#defines" statements will be processed and replaced actually source codes in this step.

**Compilation** - Using a Compiler program to convert C expanded source to assembly source code.

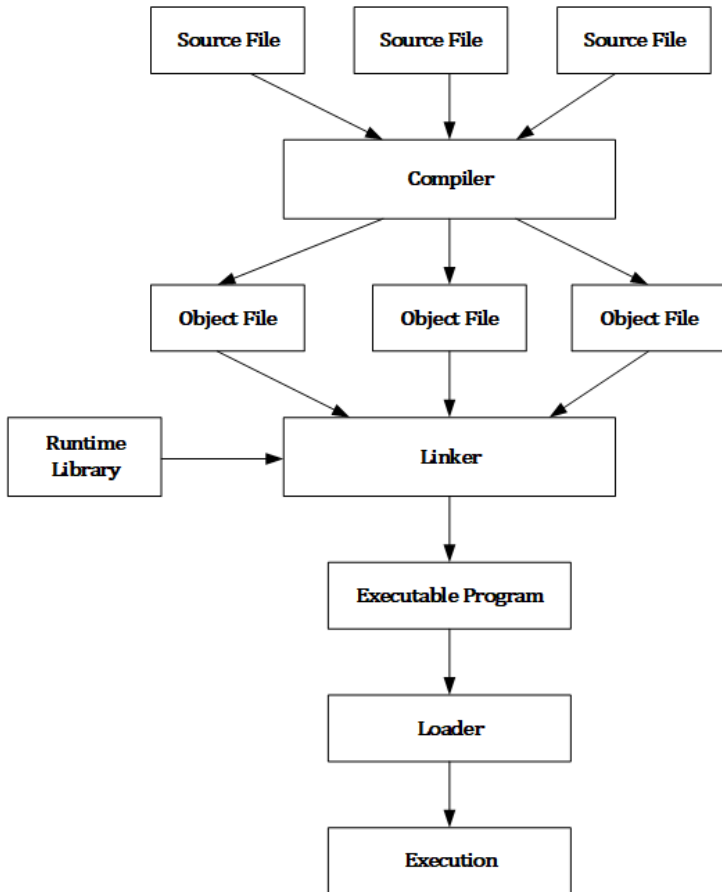
**Assembly** - Using a Assembler program to convert assembly source code to object code.

**Linking** - Using a Linker program to convert object code to executable code. Multiple units of object codes are linked to together in this step.

**Loading** - Using a Loader program to load the executable code into CPU for execution.

General compilation process is shown in Figure below:

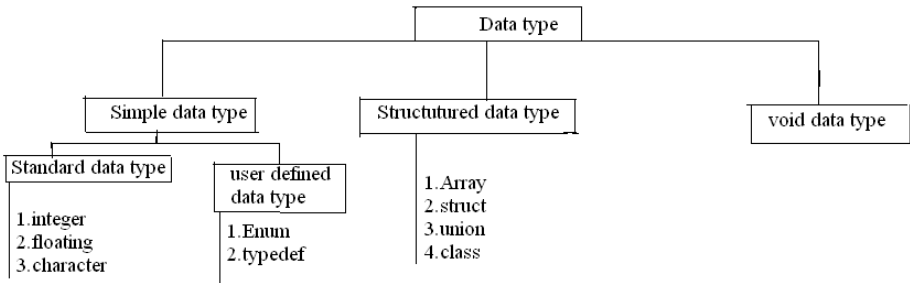




## 1.10 Primitive Data Types

**Data types in C:** Name given to set of values which have common property is known as data type. [or] A data type defines a set of values and the operations that can be performed on them are called a data type.

## C- Data types are classified as



**1. Integer data type:** Integer data types are basic data type. For example int we can assign any value which has no fractional part to int type. C offers three different integer data types they are

**Int    short int    long int**

The difference between these three integers is the number of bytes to occupy and the range of values.

Data type	Format string	Number of Bytes required	Range
int	%d	2	-32768 to 32767( $-2^{15}$ to $2^{15}-1$ )
Short int	%d or %I	2	-32768 to 32767( $-2^{15}$ to $2^{15}-1$ )
Long int	%ld	4	-2147483848 to 2147483847 ( $-2^{31}$ to $2^{31}-1$ )
Unsigned int	%u	2	0 to 65535( $0$ to $2^{16}-1$ )
Unsigned short int	%u	2	0 to 65535( $0$ to $2^{16}-1$ )
Unsigned long int	%u	4	0 to 4294967295( $0$ to $2^{32}-1$ )

**2. Float data type:** floating data types are the number which consists fractional part also. Like integers floats are divided into three types. They are

**float    double    long double**

Data type	Format string	Number of bytes	Range
Float	%f	4	3.4E-38 to 3.4E+38
Double	%lf	8	1.7E-308 to 1.7E+3.8
Long double	%lf	10	3.4E-4932 to 3.4E+4932

**3. Character data type:** A char data type can store an element of machine's character set and will occupy 1 byte.

It is of two types. They are

**Signed char**

**unsigned char**

<b>Data type</b>	<b>Format string</b>	<b>Number of bytes</b>	<b>Range</b>
Signed char	%c	1	-128 to +127
Unsigned char	%c	1	0 to 255

**User defined data type:**

1. Enumeration data type: An enumerated data type is a set of values represented by identifiers called enumeration constants.

Syntax:       enum    data type name  { const1, const2, const3,.....};

**Example:** enum dat { mon, tue, wen, thu, fri,sat, sun };

        Enum day  day1, day2, day3;

Day1, day2, day3 are enumerated data types variables. We can assign any member to day1, day2, day3 variable like

Day1=mon;

```
Day2=tue;
```

```
Day3=wen; .....
```

An enumerated data type is a user defined data type which provides away for attaching names to numbers. It increasing comprehensibility of the code. This can help in making the program listing more readable. Enumerated variables are usually used to clarify the operation of a program and make the program readable.

**2. Typedef:** The users can define an identifier that represents an existing data type by a feature known as “typedef”.(Creating a new data type is called typedef) The user defined data type identifier can later be used to declare variables.

**Syntax:** typedef type identifier;

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

**Example:**

```
#include<stdio.h>
#include<conio.h>
#define H 60
void main( )
{
```

```

typedet int hours;
hours hrs;
printf("Enter the hours");
scanf("%d",&hrs);

printf("minutes=%d",hrs*h);
printf("seconds=%d", hrs*h*h);
getch();
}

```

**Void data type:** void is an empty data type defined by the keyword void. It is used with functions. Void data type is used in three places.

a) Before the function      b) declared the void inside the function      c) both

**a) Before the function:** If we use before the function it does not return any value.

Example: void main( )

**b) Declare the void inside the function:** If we use the void inside the function, that indicates the function does not takes any argument.

**Example:** main( void)

c) **Both a and b:** If we use the void before the function and inside of the function that indicates, the function neither returns a value nor requires any argument.

**Example:** void main(void).

### **Structured data type:**

The structured data types are

Array    Structure    Union    Class

**Array:** Array is a collection of elements of similar data types in which each element is located in separate memory location.

**Structure:** The struct is keyword and used to combine variables of different data types into a single record.

**Union:** A union is same as a structure. The only difference is that all the variables will share the same memory space.

**Class:** It is defined as set of data members and member functions in a single unit is called a class.

<b>Class name</b>
<b>Data members</b>
<b>members functions</b>

## 1.11 Variables

A variable name is an identifier or symbolic name assigned to the memory location where data is stored. A variable can have only one value assigned to it at any given time during the execution of the program.

Rules regarding naming variables:

1. The variable name is an identifier, the same rules apply.
2. Meaningful names should be given so as to reflect value it is representing.

### **Syntax:**

Data type    variable1, variable2 .....

### **Example:**

```
int    I, count;
```

```
float price, salary;
```

```
char    c;
```

Where variable are declared

Variables will be declared in three basic places.

- Inside functions.
- In the definition of function parameters
- Outside of all functions.



These are

a) Local variables      b) Global variables      c) Formal variables

**a) Local variables:** variables that are declared inside a function are called local variables. Local variables exist only while the block of code in which they are declared is executing.

**Example:**

```
void fun1(void)
{
    int x;      local variables
    x=10;
}
```

**b) Global variable:** variables that are declared outside of all functions is called a global variables.

**Example:**

```
int x;      global variables
void fun1(void)
{
    x=10;
}
```

**c) Formal parameters:** If a function is use arguments, it must declare variables that will accept the values of the arguments. These variables are called the formal parameters of a function.

**Example:**

```
Void add(int y)
{
    int y=5;
    y=y+5;
    printf (“value of y=%d”, y);
}
```

**Constant and volatile variables:**

**Constant variable:** If we want that the value of a certain variable remains the same or remains unchanged during the execution of a program then, it can be done only by declaring the variable as a constant.

The keyword const is then added before the declaration. It tells the compiler that the variable is a constant.

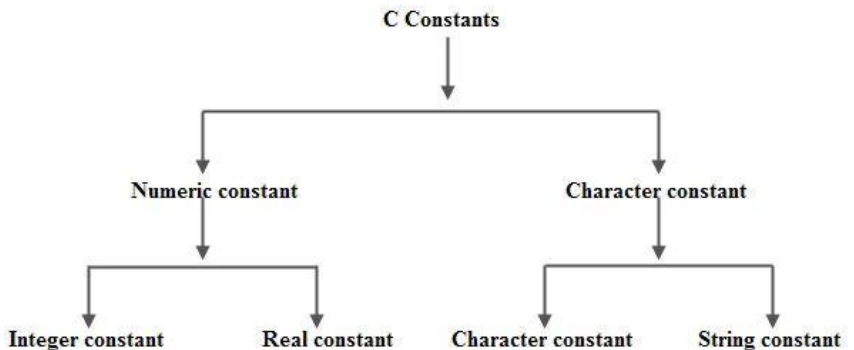
**Example:** const int m=10;

**Volatile variable:** The volatile variables are those variables that are changed at any time.

**Example:** `volatile int d;`

## 1.12 Constants

Constants refer to fixed values that do not change during program execution. They can be classified as



Numerical constants are classified into

1. Integer constants
2. Real constants

**1. Integer constants:** An integer constant is a sequence of digits without a decimal point. No commas and no blank spaces are allowed. It can be either +ve or - ve. If no sign proceeds it is assumed to be +ve. It requires minimum of two bytes.

**Example:** 10 20 +30 -15

**2. Real constants:** Real constants often known as floating point constants. These are real numbers having a decimal point or an exponential or both.

**Example:** 0.246 9.345

**Character constants:** Character constants are classified into two types.

1. Single character constants
2. String constants

**1. Single character constants:** Any character written within single quotes is called character constant. Character constants have integer values known as ASCII values.

**Example:** 'a' '8'

Escape sequence: C supports some special character constants used in output functions. They are also called character constants because they contain a black slash and a character. The complete set of escape sequence is

Char	Meaning
\a	alert (bell)
\b	back space
\f	form feed
\n	new line
\r	carriage return

<code>\t</code>	horizontal tab
<code>\v</code>	vertical tab
<code>\0</code>	NULL char
<code>\\</code>	back slash
<code>\?</code>	question mark

**2. String constants or string literals:** A string constant is a sequence of zero or more characters enclosed in double quotes. String constants are also known as string literals.

**Example:** “ welcome to hello world”

The internal representation of a string has a NULL char (`\0`) at the end. Therefore the physical storage required is one more than the number of characters in the string.



### 1.13 Operators and expressions

Operator is a symbol, which operates one or more operands with a specific meaning. C has very rich in-built in operators. An operator tells the computer to perform some specific operation.

**Example:** sum=a+b where a and b are operands and + is an operator

**Expression:** The combination of operators and operand is called an expression.

	Operators	Type
Unary Operator →	++, --	Unary Operator
Binary Operator {	+, -, *, /, %	Arithmetic Operator
	<, <=, >, >=, ==, !=	Relational Operator
	&&,   , !	Logical Operator
	&,  , <<, >>, ~, ^	Bitwise Operator
	=, +=, -=, *=, /=, %=	Assignment Operator
Ternary Operator →	?:	Ternary or Conditional Operator

### a) Arithmetic operators:

```
/* Write a c program based on arithmetic operator */
#include<stdio.h>
#include<conio.h>
void main( )
{
    printf(“%d”, 10+2);
    printf(“%d”, 10-2);
    printf(“%d”, 10*2);
}
```

```
printf(“%d”, 10/2);  
printf(“%d”, 10%2);  
getch();  
}
```

**Output:** 12 8 20 5 0

/\* write a c program to show precedence of the arithmetic operators \*/

```
#include<stdio.h>  
#include<conio.h>  
void main( )  
{  
printf(“%d”, 2+3*2);  
printf(“%d”, 6/6*3);  
printf(“%d”, 3+3/6);  
printf(“%d”, (3+3)/6);  
getch();  
}
```

**Output:** 8 3 3 1

### b) Assignment operators:

```
k=0;  
int a, b, c;  
a=10;
```

Statement with simple assignment operator	Statement with short hand operator
$a=a+1$	$a+=1$
$a=a-1$	$a-=1$
$a=a*(n+1)$	$a*=n+1$
$a=a/(n+1)$	$a/=n+1$
$a=a\%b$	$a\%=b$
<b>Table: short hand assignment operators</b>	

### Advantages of shorthand assignment operator:

- 1.What appears on the left-hand side need not be repeated and therefore it becomes easier to write.
- 2.The statement is more concise and easier to read.
- 3.The statement is more efficient.

/\* Write a C program based on the assignment operator \*/

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

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

```
void main( )
```

```
{
```

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



```
a=10;
b=c=a;
a+=b+c;
printf("a=%d b=%d c=%d", a, b, c);
getch();
}
```

**Output:** a=30 b=10 c=10

/\* Write a C program to prints a sequence of squares of numbers using shorthand operator \*/

```
#include<stdio.h>
#include<conio.h>
#define N 100
#define A 2
void main( )
{
    int a;
    a=A;
    while(a<N)
    {
        printf("%d\n", a);
        a*=a;
    }
}
```

```
getch();  
}
```

**Output:** 2 4 16

**c) Conditional operator [or] ternary operator:** The conditional operator? and: are sometimes called ternary operator. It operates on three operands, and it is a conditioned form of an if-else C statement.

Syntax: Exp1? Exp2 : Exp3

If Exp1 is true then Exp2 will be executed. If Exp1 is false then Exp3 will be executed.

**Example:** Write a c program to use the conditional operator with two statements

```
#include<stdio.h>  
#include<conio.h>  
void main( )  
{  
clrscr();  
3>2 ? printf("True") : printf("False");  
getch();  
}
```

Write a c program to check whether given num is zero or non zero using conditional operator or ternary operator.

```

#include<stdio.h>
#include<conio.h>
void main( )
{
    int num;
    clrscr();
    printf("enter the number");
    scanf("%d",&num);
    num? printf("non zero") : printf("zero");
    getch();
}

```

Write a c program to find the large value among three given values using ternary operator

```

#include<stdio.h>
#include<conio.h>
void main( )
{
    int a, b, c, max;
    clrscr();
    printf("enter the a, b, c values");
    scanf("%d%d%d", &a,&b,&c);
}

```

```

max= (a>b && a>c) ? a: ((b>c) ? b: c);
printf("max=%d", max);
getch();
}

```

**d) Relational operator:** The operators are used to compare arithmetic , logical and character expressions.

Operator	Purpose
= =	is equal to
!=	not equal
<	less than
>	grater than
<=	less than or equal
>=	greater than or equal

Write a c program show the activity of relational operators

```

#include<stdio.h>
#include<conio.h>
void main( )
{
    clrscr();
        printf("%d" , 2<3);
        printf("%d" , 2>3);
        printf("%d" , 2==3);
}

```

```

printf(“%d” , 2!=3);
printf(“%d” , 2<=3);
printf(“%d” , 2>=3);
getch();
}

```

**e ) Logical operator:** These operators are used to compare or evaluate logical and relational expressions. The result of the logical expression will be either true(1) or false (0).

Operator	Activity
&&	AND
	OR
!	Not

**AND (&&):** If both inputs are true then the output is true otherwise false.

A	B	A&&B
0	0	0
0	1	0
1	0	0
1	1	1

**OR( || ):**  If both inputs are false then the output is false otherwise true.

A	B	A&&B
---	---	------

0	0	0
0	1	1
1	0	1
1	1	1

Not (!) : If the input is true then output is false. If the input is false then output is true

A	~A
0	1
1	0

Write a c program to display truth table of AND using && operator

```
#include<stdio.h>
#include<conio.h>
void main( )
{
    clrscr();
    printf(" 0&&0=%d\n" , 0&&0);
    printf(" 0&&1=%d\n" , 0&&1);
    printf(" 1&&0=%d\n" , 1&&0);
    printf(" 1&&1=%d\n" , 1&&1);
    getch( );
}
```

Write a c program to display truth table of || (OR)

```
#include<stdio.h>
#include<conio.h>
void main( )
{
    clrscr();
        printf(" 0||0=%d\n" , 0||0);
        printf(" 0||1=%d\n" , 0||1);
        printf(" 1||0=%d\n" , 1||0);
        printf(" 1||1=%d\n" , 1||1);
        getch( );
    }
```

Write a c program to display truth table of (!) not operator

```
#include<stdio.h>
#include<conio.h>
void main( )
{
    clrscr();
        printf(" !0=%d\n" , !0);
        printf(" !1=%d\n" , !1);
        getch( );
    }
```

**f) Bitwise operator:** These operators are used to operate with bits of data for complementing, testing, setting or shifting the bits of data in a byte or word.

<b>Operator</b>	<b>Operator name</b>	<b>Purpose</b>
~	complement	for 1's complement
>>	right shift	to move bits to right
<<	left shift	to move bits to left
&	bitwise AND	to reset (0) / compare bits
!	bitwise OR	to reser(1)/ compare bits
^	bitwise exclusive OR (XOR)	to clear the bits

**Note:** Bitwise operators work only with char and int data types and not useful for float, double, void or complex data types.

Write a c program for ~ (complement operator)

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



```
}
```

Write a c program for bitwise AND operator (&)

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

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

```
void main( )
```

```
{
```

```
    char a=10;
```

```
    clrscr();
```

```
        printf(“ %d\n” , a&2);
```

```
        getch();
```

```
}
```

Output: 2

Hint: Binary form of 10 = 0 0 0 1 0 1 0

Binary form of 2 = 0 0 0 0 0 0 1 0

AND operation = 0 0 0 0 0 0 1 0

Write a c program for bitwise OR operator (|)

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

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

```
void main( )
```

```
{
```

```

char a=10;
clrscr();
    printf(“ %d\n” , a/2);
    getch();
    }

```

Output: 10

**Hint:** Binary form of 10 = 0 0 0 0 1 0 1 0  
 Binary form of 2 = 0 0 0 0 0 0 1 0  
 OR operation = 0 0 0 0 1 0 1 0

Write a c program for bitwise XOR operator ( ^ )

```

#include<stdio.h>
#include<conio.h>
void main( )
{
    char a=10;
    clrscr();
        printf(“ %d\n” , a^2);
        getch();
    }

```

Write a c program to shift input data by two bits right (>>)

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

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

```
void main( )
```

```
{
```

```
    int x, y;
```

```
    clrscr();
```

```
    printf("enter the number ");
```

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

```
    y=x>>2;
```

```
    printf(" the right shifted data is %d", y);
```

```
    getch( );
```

```
}
```

**Formula:-**  $y=n/2^s$  where n is number and s is number of position to be shifted.

Write a c program to shift input data by two bits right (<<)

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

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

```
void main( )
```

```
{
```

```
    int x, y;
```

```
    clrscr();
```

```

printf("enter the number ");
scanf("%d",&x);
y=x<<2;
printf(" the right shifted data is %d", y);
getch();
}

```

Formula:-  $y=n*2^s$  where n is number and s is number of position to be shifted.

**g) Increment and decrement operator:** C has tow useful operators

- 1) ++ (increment operator)
- 2) – (decrement operator)

The ++ operator adds 1 to its operand. The – operator subtracts 1 from its operand.

These operators may be either pre increment/or pre decrement or post increment /post decrement.

Write a c program to show the effect of increment operator as a suffix.

```

#include<stdio.h>
#include<conio.h>
void main( )
{

```

```
int a, z, x=10, y=20;
clrscr();
z= x*y++;
a=x*y;
printf(“ %d %d”, z,a);
getch();
}
```

Output: z=200 a=210

Note: z=x\*y++ means first y value is assigned, later y will be incremented so z=200 and a=210

Write a c program to show the effect of increment operator as a prefix.

```
#include<stdio.h>
#include<conio.h>
void main( )
{
int a, z, x=10, y=20;
clrscr();
z= x* ++y;
a=x*y;
printf(“ %d %d”, z,a);
```

```
        getch();
    }
```

Output: z=210 a=210

Write a c program to perform increment and decrement operator

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

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

```
void main( )
```

```
{
```

```
    int x=6, y=4, z;
```

```
    clrscr();
```

```
    z= ++x + y-- + x++ - x-- + y++ + --y +
```

```
    ++x;
```

```
    printf("y= %d    z= %d    x= %d" y, z, x);
```

```
    getch();
```

```
}
```

**H) Special operators:** The special operators are

- sizeof ( ) operator
- Comma operator ( , )
- Address of operator (&)
- Value at address operator (\*)

**1) sizeof ( ) operator:** the sizeof operator is used to obtain the size of a variable which occupies in system's memory. The sizeof() return value (in bytes) is the size of a variable or type with in the parenthesis.

**Syntax:** sizeof ( object);

Where object may be any data type, variable or expression.

Example: int n1;  
n1= sizeof(int);

Write a c program on sizeof( ) operator

```
#include<stdio.h>
#include<conio.h>
void main( )
{
    int x;
    float y;
    clrscr();
    printf("\n sizeof(x) =%d bytes", size(x));
    printf("\n sizeof(y) =%d bytes", size(y));
    printf("\n address of x =% u and y=%u", &x,&y);
    getch();
}
```

**2) Comma operator:** The comma operator separates the expressions

Syntax: exp1, exp2,.....expn;

Example: #include<stdio.h>

#include<conio.h>

void main( )

{

printf(“addition=%d\n sub=%d”, 2+3, 5-4);

getch( );

}

**3) Address of operator (&):** This operator is used to find the address of a variable of any data type.

Example: m=&n; here address of n is assigned to m. This m is not a ordinary variable, it is a variable which holds the address of the other variable.

**4) Value at address (\*):** This operator is used to find the value at a particular memory location. This operator is also called as indirection operator or dereferencing operator.



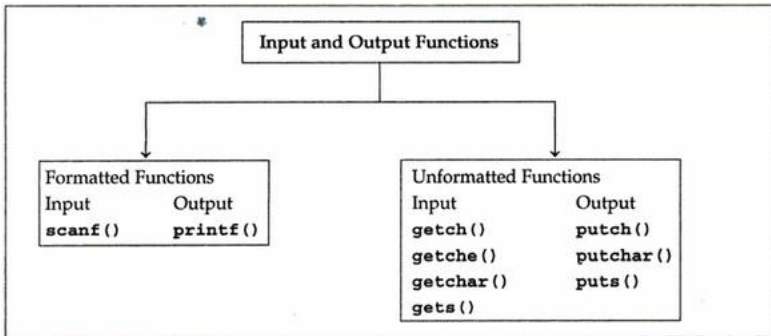
## 1.14 Basic Input and Output

Reading the data from the input device and displaying the results on the screen, are the two main tasks of any program. To perform these tasks user friendly C has a number of input and output functions. When a program needs data, it takes the data through the input functions and sends results obtained through the output functions.

There are number of I/O functions in C based on the data types. The input/output functions are classified into two types

**1) Formatted functions:** The formatted input/output functions read and write all types of data values. They require conversion symbol to identify the data type.

**2) Unformatted functions:** The unformatted input/output functions only work with the character data type. They do not require conversion symbol for identification of data types because they work only with character data type.



## Formatted functions

### a) The printf ( ) statement

The printf ( ) function prints all types of data values to the console. It requires conversion symbol and variable names to print the data. The conversion symbol and variable names should be same in number. The example of print ( ) statement is given below.

#### **Example**

```

main()
{
    int x=2;
    float y=2.2;
    char z='C';
    printf("%d %f %c",x,y,z);
}

```

**OUTPUT:**  
2 2.2000 C

## b) The scanf ( ) statement

The scanf ( ) statement reads all types of data values. It is used for runtime assignment of variables. The scanf ( ) statement also requires conversion symbol to identify the data to be read during the execution of a program.

### Syntax

Scanf (“%d %f %c”, &a, &b, &c);

```
4.1 Write a program to show the effect of mismatch of data types.

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

void main()
{
    int a;
    clrscr();
    printf("Enter value of 'A': ");
    scanf ("%c",&a);
    printf ("A=%c",a);
}

OUTPUT:
Enter value of 'A' : 8
A=8
```

## Unformatted functions

C has two types of I/O functions.

- Character I/O
- String I/O

### a) Character I/O

The **getchar()** and **putchar()** Functions

The **int getchar(void)** This function reads character type data from the standard input. It reads one character at a time till the user presses the enter key.

The **int putchar(int c)** function puts the passed character on the screen and returns the same character. This function puts only single character at a time.

```
#include <stdio.h>
int main( )
{
    int c;
    printf( "Enter a value :");
    c = getchar( );
    printf( "\nYou entered: ");
    putchar( c );
    return 0;
}
```

When the above code is compiled and executed, it waits for you to input some text. When you enter a text and press enter, then the program proceeds and reads only a single character and displays it as follows –

```
$/a.out
```

```
Enter a value : this is test
```

You entered: t

### **The getch ( ) and getche( )**

These functions read any alphanumeric characters from the standard input device. The character entered is not displayed by getch ( ) function.

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

void main()
{
  clrscr();
  printf("Enter any two alphabetic " );
  getche();
  getch();
}
OUTPUT:
Enter any two alphabetic A
```

### **The putch( )**

This function prints any alphanumeric character taken by the standard input device.

```
main ()
{
char ch;
clrscr ();
printf("Press any key to continue ");
ch=getch();
printf ("\n You Pressed : ");
putch(ch);
}

OUTPUT:

Press any key to continue
You Pressed : 9
```

## b) String I/O

The gets() and puts() Functions

The gets()

This function is used for accepting any string through stdin(keyboard) until enter key is pressed. The header file stdio.h is needed for implementing the above function.

The puts()

This function prints the string or character array.

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

```
int main()
```

```
{
```

```
    char str[100];
```

```
    printf( "Enter a value :");
```

```
gets( str );  
printf( "\nYou entered: ");  
puts( str );  
return 0;  
}
```

When the above code is compiled and executed, it waits for you to input some text. When you enter a text and press enter, then the program proceeds and reads the complete line till end, and displays it as follows –

```
$/a.out
```

```
Enter a value : this is test
```

```
You entered: this is test
```

### **1.15 Type conversion**

Sometimes the programmer needs the result in certain data type. For example, division of 5 with 2 should return float value. Practically the compiler always returns integer value because both the arguments are of integer data type.

The conversion of value of one data type to another data type is called type casting.

**Syntax :**            (data type) exp;

They are two types of type conversion

- Implicitly type conversion
- Explicitly type conversion

**1. Implicitly type conversion:** The automatic conversion from one data type into another data type is called a implicitly type conversion.

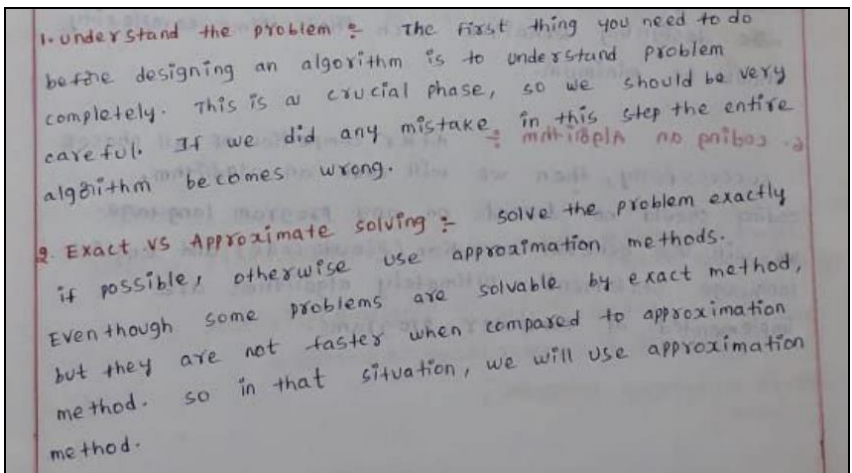
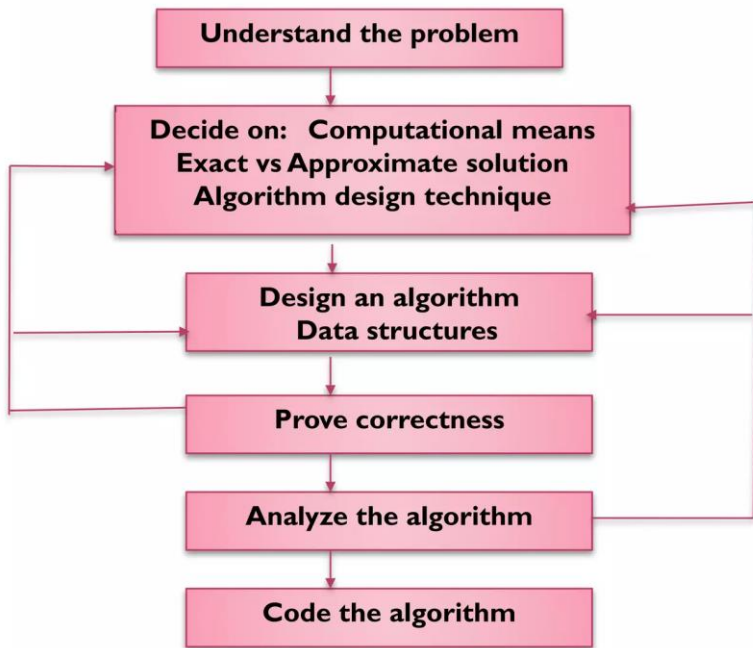
**2. Explicitly type conversion:** The force to convert from one data type into another data type is called a explicitly type conversion.

Example:

```
#include<stdio.h>
#include<conio.h>
void main()
{
    clrscr();
    printf("two intergers(5 and 2) %d", 5/2);
    printf("two integers(5 and 2) %f", (float) 5/2);
    getch();
}
```



## 1.16 Problem Solving Techniques: Algorithmic approach



3. Algorithm design techniques :- In this we will use different design techniques like

- a) Divide and conquer
- b) Greedy method
- c) Dynamic programming
- d) Backtracking
- e) Branch and bound
- f) Brute force
- g) Decrease and conquer
- h) Transform and conquer
- i) Space and Time trade-offs

depending upon the problem, we will use suitable design method.

4. prove correctness :- once an algorithm has been specified, next we have to prove its correctness. usually testing is used for providing correctness.

5. Analyze an Algorithm :- Analysing the algorithm means studying the algorithm behaviour is calculating the time complexity and space complexity. If the time complexity of algorithm is more, then we will use one more designing technique such that time complexity should be minimum.

6. coding an Algorithm :- After completion of all phases successfully, then we will code an algorithm. coding should not depend on any program language. we will use general notation (Pseudo code) and English language statement. Ultimately algorithms are implemented as computer programs.

## 1.17 Problem solving strategies: Top-down approach, Bottom-up approach

S.No.	Top-Down Approach	Bottom-Up Approach
1.	In this approach, the problem is broken down into smaller parts.	In this approach, the smaller problems are solved.
2.	It is generally used by structured programming languages such as C, COBOL, FORTRAN, etc.	It is generally used with object oriented programming paradigm such as C++, Java, Python, etc.
3.	It is generally used with documentation of module and debugging code.	It is generally used in testing modules.
4.	It does not require communication between modules.	It requires relatively more communication between modules.
5.	It contains redundant information.	It does not contain redundant information.
6.	Decomposition approach is used here.	Composition approach is used here.
7.	The implementation depends on the programming language and platform.	Data encapsulation and data hiding is implemented in this approach.

## 1.18 Time and space complexities of algorithms.

Performance analysis of an algorithm is done to understand how efficient that algorithm is compared to another algorithm that solves the same computational problem. Choosing efficient algorithms means computer programmers can write better and efficient programs.

A computer resource is memory and CPU time and performance analysis revolves around these two resources.

Two ways to evaluate an algorithm is listed below.

- Space requirement
- Computation time

The space requirement is related to memory resource needed by the algorithm to solve a computational problem to completion. The program source code has many types of variables and their memory requirements are different. So, you can divide the space requirement into two parts.

### **Fixed Variables**

The fixed part of the program are the instructions, simple variables, constants that does not need much memory and they do not change during execution. They are free from the characteristics of an instance of the computational problem.

### **Dynamic Variables**

The variables depends on input size, pointers that refers to other variables dynamically, stack space for recursion are some example. This type of memory requirement changes with instance of the problem and depends on the instance characteristics. It is given by the following equation.

Instance characteristics means that it cannot be determined unless the instance of a problem is running which is related to dynamic memory space. The input size usually has the control over instance of a computational problem.

### **Time Complexity**

The time complexity is the amount of time required to run the program to completion. It is given by following.

### **Program Execution in Steps**

The computer executes a program in steps and each step has a time cost associated with it. It means that a step could be done in finite amount of time. See the table below.

<b>Program Step</b>	<b>Step Value</b>	<b>Description</b>
Comments	zero	comments are not executed
Assignments	1 step	can be done in constant time
Arithmetic Operations	1 step	can be done in constant time
Loops	Count Steps	if loop runs n times, count step as n+1 and any assignment inside loop is counted as n step.
Flow Control	1 step	only take account of part that was executed.

You can count the number of steps an algorithm performed using this technique. For example, consider the following example.

Operations	S/e	Frequency	Total Steps
Algorithm Sum(a, n)	0	-	0
{	0	-	0
s:= 0;	1	1	1
for i:= 1 to n do	1	n + 1	n + 1
s:= s + a[i];	1	n	n
return s;	1	1	1
}	0	-	0
		Total	2n + 3

## UNIT-II

### **Control Structures**

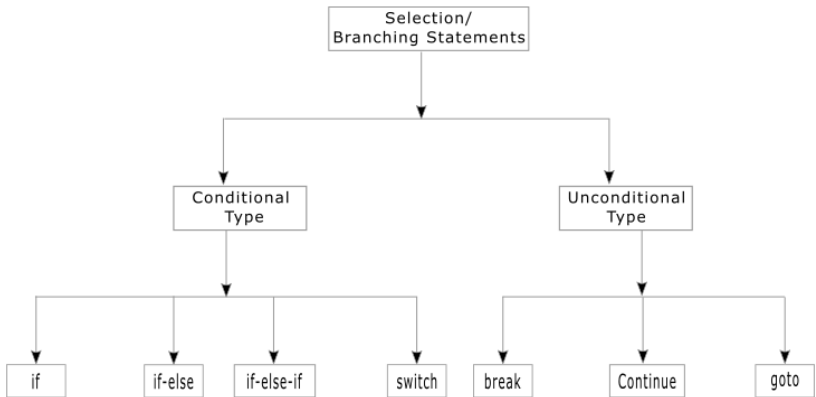
Simple sequential programs Conditional Statements (if, if-else, switch), Loops (for, while, do while) Break and Continue.

#### **2.1 Decision Statements (Control Statements)**

Introduction: In any program statements are normally executed. We have number of situations repeat a group of statements until certain specified condition or change the order of execution of statements. **The C language supports the control statements as listed below**

1. The simple if statement
2. The if-else statement
3. Nested if-else statement
4. The if-else-if ladder statement
5. The switch( ) case statement
6. The nested switch( ) case statement

# Flowchart Control Statements

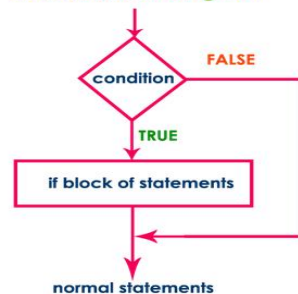


**1. The simple if statement:** The if statement is used to specify conditional execution of program statements or a group of statements enclosed in braces.

## Syntax

```
if ( condition )  
{  
    ....  
    block of statements;  
    ....  
}
```

## Execution flow diagram



The statement is executed only when the condition is true. In case the condition is false the compiler skips the lines



within a pair of parenthesis. The conditional statements should not be terminated with semi colons.

**Example 1: Write a c program to check whether the entered number is less than 10? If yes, display the same.**

```
#include<stdio.h>
#include<conio.h>
void main( )
{
    int n;
    clrscr();
    printf("Enter the number");
    scanf("%d",&n);
    if(n<10)
    printf("\n number is less than 10");
    sleep(2);
    getch();
}
```

**Example 2: Write a c program to check whether the candidates age is greater than 17 or not. If yes display message "Eligible for voting".**

```
#include<stdio.h>
#include<conio.h>
void main( )
{
    int age;
    clrscr();
    printf("Enter the age");
    scanf("%d",&age);
    if(age>17)
    printf("\nEligible for voting");
    getch();
}
```

**Example 3: Write a c program to use curly brace in the if block. Enter only the three numbers and calculate their sum and multiplication.**

```
#include<stdio.h>
#include<conio.h>
void main( )
{
    int a,b,c,x;
    clrscr();
```

```

printf("Enter three numbers");
scanf("%d%d%d",&a,&b,&c);
if(x==3)
printf("\n sum=%d", a+b+c);
printf("\n multiplication=%d", a*b*c);
getch();
}

```

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

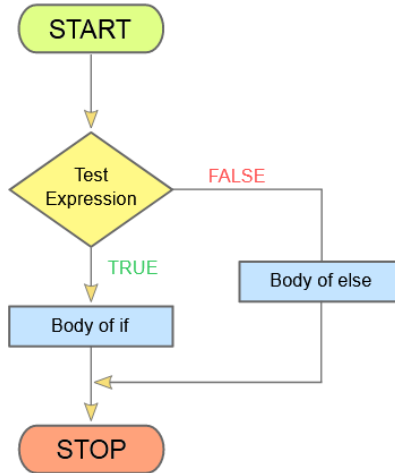
**Syntax:**

```

:   if (condition)
{
Statement1;
Statement2;      If block (True block)
}
else
{
Statement3;
Statement4;      else block (False block)
}

```

}



**Example 1: Read the values of a, b,c through the keyboard add them and after addition check if it is in the range of 100 and 200 or not. Print separate message for each**

```
#include<stdio.h>
#include<conio.h>
void main()
{
    int a,b,c,d;
    clrscr();
    printf("Enter the three numbers");
```

```

scanf("%d%d%d",&a,&b,&c);
d=a+b+c;
if(d<=200&&d>=100)
printf("\n sum is %d which is in between 100 and 200", d);
else
printf("\n sum is %d which is outof range ",d);
getch();
}

```

**Example 2: Write a c program to calculate the salary of medical representative bases on the sales. Bonus and incentive to be offered to him will be based on total sales. If the sales exceeds Rs 100000/- follow the particulars of table(1) otherwise table(2).**

Table 1

Basic= 3000

H.R.A=20% of basic

D.A=110% of basic

Conveyance=500

Incentive=10% of sales

Bonus=500

Table 2

Basic=3000

H.R.A =20% of basic

D.A=110% of basic

conveyance=500

Incentive=5% of sales

Bonus=200

```

#include<stdio.h>
#include<conio.h>
void main()
{
    float bs,hra,da,cv,incentive,bonus,sales,ts
    clrscr();
    printf("enter the toatal sales");
    scanf("%d",&sales)l
    if(sales>=100000)
    {
        bs=3000;
        hra=20*bs/100;
        da=110*bs/100;
        cv=500;
        incentive=sales*10/100;
        bonus=500;
    }
    else
    {
        bs=3000;
        hra=20*bs/100;
        da=110*bs/100;
    }
}

```

```

cv=500;
incentive=sales*5/100;
bonus=200;
}
ts=bs+hra+da+cv+incentive+bonus;
printf("\n total sales=%f",sales);
printf("\n basic salary=%f",bs);
printf("\n H.R.A=%f",hra);
printf("\n D.A=%f",da);
printf("\n conveyance=%f",cv);
printf("\n Bonus=%f",bonus);
printf("\n gross salary=%f",ts);
getch();
}

```

**3.The if-else-if ladder statement:** In this kind of statements number of logical conditions are executing various statements here, if any logical condition is true the compiler executes the block followed by if condition otherwise it skips and executes else block. In if-else statement else block is executed by default after failure of condition.

## **Syntax:**

```
if(condition)
```

```
{
```

```
    statement1;
```

```
    statment2;
```

```
}
```

```
else if(condition)
```

```
{
```

```
    statement3;
```

```
    statement4;
```

```
}
```

```
else
```

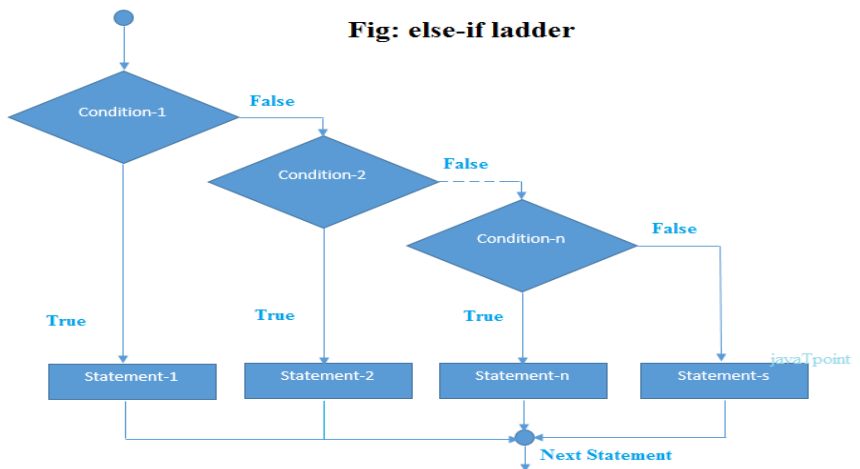
```
{
```

```
    statement5;
```

```
    statement6;
```

```
}
```





**Example 1: Write a c program to calculate electricity bill. Read the starting and ending meter reading. The charges are as follows.**

No. of units consumed	Rate in (RS)
200-500	3.50
100-200	2.50
Less than 100	1.50

```
#include<stdio.h>
#include<conio.h>
void main()
{
    int intial, final, units;
    float price;
```

```

clrscr();
printf("\n enter the intial and final readings");
scanf("%d%d",&initial, &final);
untis=final-initial;
if(units>=200&&units<=500)
    price=uints*3.50;
else if(units>=100&&units<=199)
    price=uints*2.50;
else if(units<100)
    price=uints*1.50;
printf("units=%d price=%f", uints,price);
getch();
}

```

**Example 2 : Write a c program to find the average of six subjects and display the result as follows.**

Average	Result
>34 and <50	Third class
>49 and <60	second class
>60 and <75	First class

If marks in any subject less than 35 Fail

```

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

```

```

void main()
{
    int sum=0,a,b,c,d,e,f;
    float avg;
    clrscr();
    printf("Enter the six subjects marks");
    scanf("%d%d%d%d%d%d",&a,&b,&c,&d,&e,&f);
    sum=a+b+c+d+e+f;
    avg=sum/6;
    printf("sum=%d avg=%f",sum,avg);
    if(a<35||b<35||c<35||d<35||e<35||f<35)
    {
        printf("\n result is fail");
        exit(0);
    }
    if(avg>34&&avg<50)
    printf("\n result is third class");
    else if(avg>49&&avg<60)
    printf("\n result is second class");
    else if(avf>60&&avg<75)
    printf("\n result is first class");
    else if(avg>75&&avg<100)

```

```
printf("\n result is distinction");  
getch();  
}
```

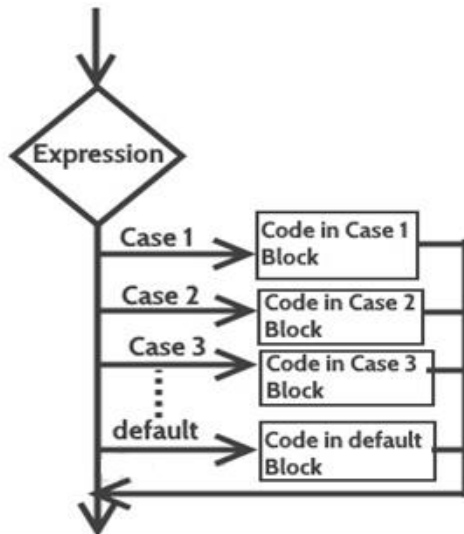
**4. The switch ( ) case statement:** The switch statement is a multi-way branch statement. In the program if there is a possibility to make a choice from a number of options, this structured selection is useful. The switch statement requires only one argument of any data type. Which is checked with number of case options. The switch statement evaluates expressions and then looks for its value among the case constants. If the value matches with case constants, this particular case statement is executed. If not default is executed.

Here switch, case and break, default are keywords. Every case statement is terminated with semicolon. The break statement is used to exit from the current case structure.

Syntax:

```
switch(variable or expression)  
{  
    case constant1:  
        statement;
```

```
        break;
case constant2:
    statement;
    break;
.
.
.
.
default: statement;
}
```



**Example: Write a c program to provide multiple functions such as 1.addition 2.substraction 3. Multiplication 4. Division**

```
#include<stdio.h>
#include<conio.h>
void main()
{
    int a,b,c,ch;
    clrscr();
    printf("-----");
    printf("\n\t Menu");
    printf("-----");
    printf("\n\t 1.Addtion");
    printf("\n\t 2.substraction");
    printf("\n\t 3.multiplication");
    printf("\n\t 4.division");
    printf("enter the a,b values");
    scanf("%d%d",&a,&b);
    printf("enter the choice");
    scanf("%d",&ch);
    switch(ch)
    {
```

```

case 1: c=a+b;
        printf("sum=%d",c);
        break;
case 2: c=a-b;
        printf("sub=%d",c);
        break;

case 3: c=a*b;
        printf("multi=%d",c);
        break;
case 4: c=a/b;
        printf("division=%d",c);
        break;
default: printf("invalid choice");
}
getch();
}

```

**5. Nested switch ( ) case:** C supports the nesting of switch ( ) case. The inner switch can be part of an outer switch. The inner and the outer switch case constants may be the same. No conflict arises even if they are same.

Example: Write a c program to demonstrated nested switch(  
) case statement

```
#include<stdio.h>
#include<conio.h>
void main()
{
    int x;
    clrscr();
    printf("enter a number");
    scanf("%d",&x);
    switch(x)
    {
        case 0:printf("number is even");
            break;
        case 1:printf("number is odd");
            break;
        default:
            y=x%2;
            switch(y)
            {
                case 0: printf("even");
                    break;
```

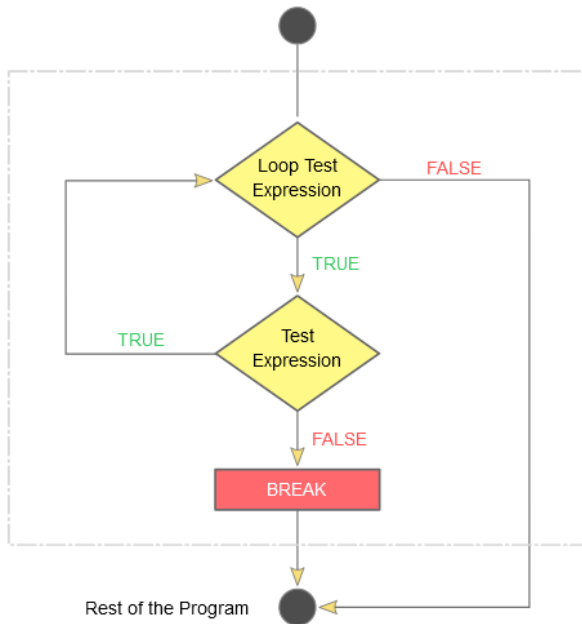


```
        default: printf("odd");
    }
}
getch();
}
```

**The break statement:** The keyword break allows the programmers to terminate the loop. The break skips from the loop or block in which it is defined.

### Break Statement Program

Let us write a C program to demonstrate break statement.



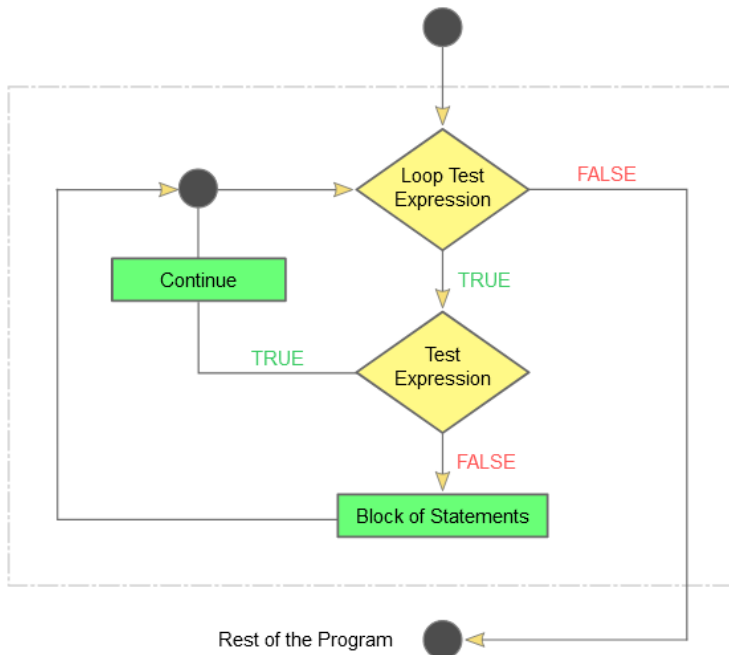
```
#include <stdio.h>
int main()
{
int a;
for(a = 1;a <= 5;a++)
{
if(a == 4)
{
break;
}
printf("%d ", a);
}
return 0;
}
```

## Output



1 2 3

**The continue statement:** The continue statement is exactly opposite to break. The continue statement is used for continuing next iteration of loop statements. When it occurs in the loop it does not terminate, but it skips the statements after this statement.



## Continue Statement Program

Let us write a C program to demonstrate continue statement

```
#include <stdio.h>
int main()
{
int a;
for(a = 1;a <= 5; a++)
{
if(a == 4)
{
continue;
}
printf("%d", a);
}
return 0;
}
```

## Output



1 2 3 5

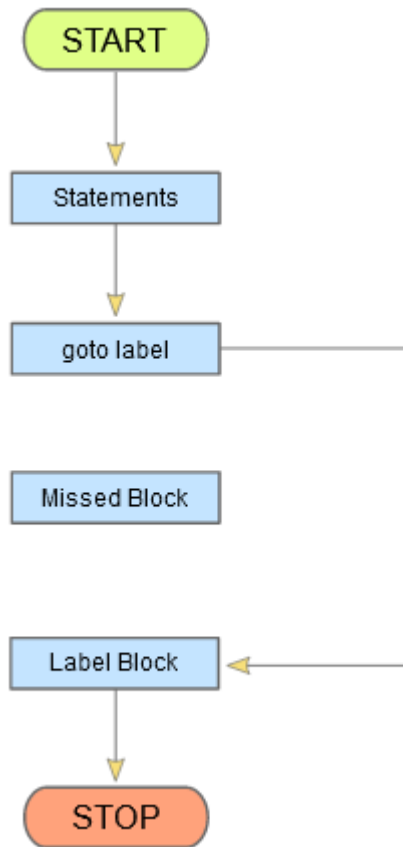
The goto statement: This statement does not require any condition. This statement passes control anywhere in the program. i.e; control is transferred to another part of the program without testing any condition.

Why goto statement

C programming introduces a goto statement by which you can jump directly to a line of code within the same file.

Syntax: `goto label;`

Where the label name must start with any character.



**Example 1: write a c program to find the entered number is even or odd using goto statement.**

```
#include<stdio.h>
#include<conio.h>
void main()
{
    int x;
    clrscr();
    printf("enter a number");
    scanf("%d",&x);
    if(x%2==0)
        goto even;
    else
        goto odd;

even: printf("\n %d is a even number");
        return;

odd: printf("\n %d is odd number");
        getch();
}
```

**Example2: write a c program to check whether the entered year is a leap year or not use goto statement.**

```
#include<stdio.h>
#include<conio.h>
void main()
{
    int leap;
    clrscr();
    printf("enter the year");
    scanf("%d",&leap);
    if(leap%4==0)
        goto leap;
    else
        goto nonleap;

leap: printf("\n %d is a lep year");
        return;
nonleap: printf("\n %d is not a leap year");
        getch();
}
```

## Difference between break and continue statement

### Break

1. Exist from current block or loop
2. Control passes to next statement
3. Terminates the program

### Continue

1. Loop takes next iteration
2. Control passes at the beginning of loop
3. Never terminates the program.

## Iterative Statements (Loop control statements)

Introduction: Many tasks are needed to be done with the help of a computer and they are repetitive in nature. For example , the salary of laborers of a factory is calculated by the formula (No.of hours\* wage rate). This calculation will be performed by an accountant for each worker every month. Such type of repetitive actions can be easily done using a program that has a loop into the solution of the problem.

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

Steps in loop:

- 1) **Loop variable:** It is a variable used in the loop.
- 2) **Initialization:** It is the first step in which starting and final value is assigned to the loop variable. Each time the updated value is checked by the loop itself.



**3) Increment/decrement:** It is the numerical value added or subtracted to the variable in each round of the loop.

The loop statements are

1. for loop
2. Nested for loop
3. The while loop
4. The do-while loop

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

**Syntax:**

```
for(initialization; condition; increment/decrement)
{
    Statement1;
    Statement2;
}
```

**Initialization:** The initial value is executed only once.

**Condition:** The condition is a relational expression that determines the number of iterations desired or it determines when to exit from the loop. The for loop continues to

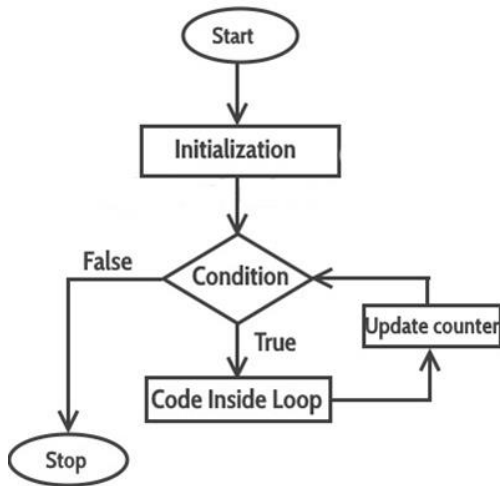
execute as long as conditional test is satisfied. When the condition becomes false the control of the program exists from the body of the for loop and executes next statement after the body of the loop.

### Various formats of for loop:

Syntax	Output	Remarks
for (; ;)	Infinite loop	No arguments
for(a=0;a<=20)	Infinite loop	'a' is neither incremented nor decrement.
for(a=0;a<=10;a++)	Display value from 0 to 10	'a' is increased from 0 to 10.
for(a=10;a>=0;a--)	Display value from 10 to 0	'a' is decreased from 10 to 0

### Syntax

```
for(exp1;exp2;exp3)  
Statement;
```



**Example1: write a c program to find the factorial of a given number**

```

#include<stdio.h>
#include<condio.h>
void main()
{
  int n, i, fact=1;
  printf("Enter the number");
  scanf("%d",&n);
  for(i=1;i<=n;i++)
  {
    fact=fact*i;
  }
}
  
```

```
    }  
    printf("The factorial of a given number is%d", fact);  
    getch();  
}
```

**Example 2: write a c program to display numbers from 1 to 10**

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

**Example3: Write a c program to find the distance travelled by a vehicle( $ut + \frac{1}{2}at^2$ )**

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

```

{
    int n,i,t;
    float s;
    clrscr();
    printf("how many times do you want repeat the process");
    scanf("%d",&n);
    for(i=1;i<=n;i++)
    {
        printf("Enter the u,a,t values");
        scanf("%d%d%d",&u,&a,&t);
        s=u*t+a*t*t/2.0
        printf("The distance travelled by a vehicle is %f", s);
    }
    getch();
}

```

**2. Nested for loop:** In nested for loops one or more statements are included in the body of the loop. The number of interactions in this type of structure will be equal to the number of interactions in the outer loop multiplied by the number of interactions in the inner loop.

**Example: Write a C program to illustrate an example based on nested for loops**

```
#include<stdio.h>
#include<conio.h>
void main()
{
    int i,j;
    clrscr();
    for(i=1;i<=3;i++)
    {
        for(j=1;j<=2;j++)
        {
            printf("%d", i*j);
        }
    }
    getch();
}
```

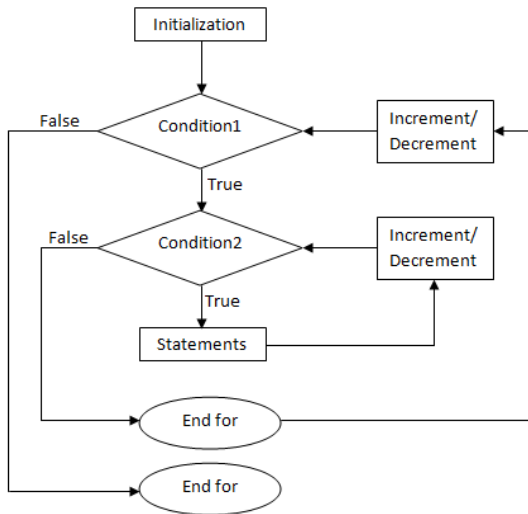


Fig: Flowchart for nested for loop

### 3. The while loop

**Syntax:** while (condition)

```

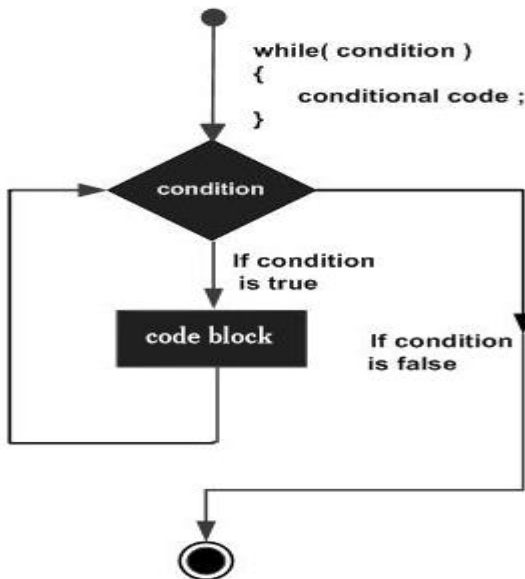
{
    Body of the loop;
}
  
```

1. The test condition is evaluated and if it is true, the body of the loop is executed.
2. On execution of the body, test condition is repetitively checked and if it is true the body is executed.

3. The process of execution of the body will be continuing till the test condition becomes false.

4. The control is transferred out of the loop.

The block of the loop may contain a single statement or a number of statements. The same block can be repeated.



**Example1:** Write a c program to print the string “welcome” 9 times using while loop

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

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

```
void main()
```



```
{
int x=1;
while(x<10)
{
printf("\n welcome");
x++;
}
getch();
}
```

**Example2:** Write a c program to add 10 consecutive numbers starting from 1 use the while loop.

```
#include<stdio.h>
#include<conio.h>
void main()
{
int a=1,sum=0;
clrscr();
while(a<=10)
{
printf("%2d",a);
sum=sum+a;
```

```
    a++;  
    }  
    printf("sum of the 10 numbers is %d", sum);  
    getch();  
}
```

**Example 3:** Write a c program to find the factorial of a given number by using while loop

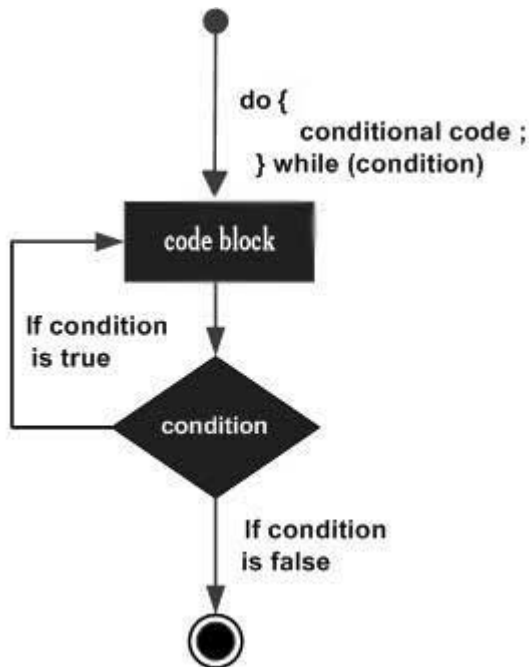
```
#include<stdio.h>  
#include<conio.h>  
void main()  
{  
    int n, i=1, fact=1;  
    printf("Enter the number");  
    scanf("%d",&n);  
    while(i<=n)  
    {  
        fact=fact*i;  
        i++;  
    }  
    printf("the factorial of a given number is%d",fact);  
    getch();  
}
```

#### 4. do-while loop

A do...while loop is similar to a while loop, except the fact that it is guaranteed to execute at least one time.

Syntax:

```
do
{
    Statements;
}
While (condition);
```



**Example 1:** Write a c program to find the factorial of a given number by using do-while

```
#include<stdio.h>
#include<conio.h>
void main()
{
    int n, i=1, fact=1;
    printf("Enter the number");
    scanf("%d",&n);
    do
    {
        fact=fact*i;
        i++;
    }
    while(i<=n);
    printf("the factorial of a given number is%d",fact);
    getch();
}
```

## Difference between while and do-while

While	Do-while
In the while loops the condition is tested following the statement and then the body gets executed.	The do-while the condition is checked at the end of the loop.
	The do-while loop will execute at least one time even if the condition is false.

### 5. The while loop within the do-while loop

#### Syntax:

```
do while(condition)
```

```
{
```

```
    Statements;
```

```
}
```

```
while(condition);
```

Example: Write a C program to use while statement in do-while and print values from 1 to 5

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

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

```
void main()
```

```
{
```

```
int x=0;
clrscr();
do while(x<5)
{
    x++;
    printf("%d",x);
}
while(x<1);
getch(); }
```

**Output:**      1          2          3          4          5

## UNIT-III

**Arrays and Strings:** Arrays indexing, memory model, programs with array of integers, two dimensional arrays, Introduction to Strings.

### **3.1 Introduction**

Consider the following example

```
main()
{
int a=2;
    a=4;
printf(“%d” ,a);
getch();
}
```

**Output:** 4.

In the above example the value of ‘a’ printed is 4. 2 is assigned to ‘a’ before assigning 4 to it. When we assign 4 to ‘a’ then the value stored in ‘a’ is replaced with the new value. Hence ordinary variables are capable of storing one value at a time. But the array variables are able to store more than one value at a time.

## **Definition of Array**

An array is defined as a set of homogeneous data items of the same type that share a common name.

### **Element:**

The individual values in the array are called as elements.

### **Index or Subscript:**

Each array element is referred by specifying the array name, followed by a number within square braces referred as an index or subscript.

**Note:** Arrays are static data structures.

## **3.2 Declaration of array**

The declaration of array is as follows.

### **Syntax:**

<b>Storage type Data type array name [size];</b>
--

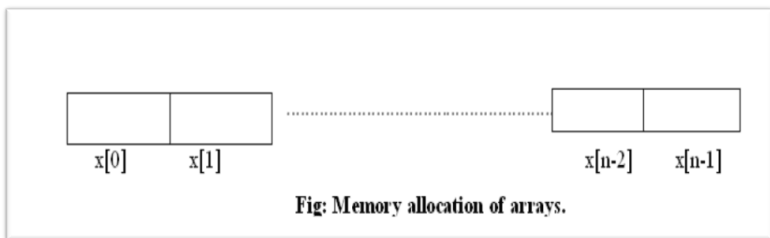
Where storage type may be either auto or register or static or extern. The storage type while declaring an array is optional. The data type specifies the type of element that will be stored in the array. The size is a +ve integer constant. Indicating the maximum number of elements that can be stored in the array.



### Example:

```
int days[31];  
char book[50];  
float real[10];
```

In an n element array the array elements are stored in  $x[0], x[1], \dots, x[n-2], x[n-1]$  as shown in fig.



### 3.3 Initialization of Arrays

#### Syntax:

<b>Storage type</b>	<b>Data type</b>	<b>array name[size]= { list of values };</b>
---------------------	------------------	--

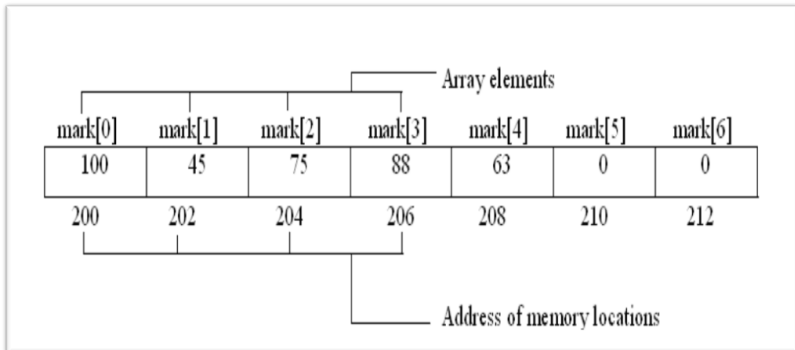
The lists of values are separated by commas used for initializing the array.

#### Example:

```
int mark [7] = { 100,45,75,88,63 };  
char name[8] = { 's','u','d','e','r',' ','10' };  
float amount [ ] = { 12.3, 234.56, 34.56, 5768.90 };
```

In the third example the size of the array is omitted. In such cases, the compiler allocates enough memory space for all

the elements given for initialization. Hence the size of the array amount [] will be 4. Fig: Shows the memory occupied by the array mark in the first example.



### 3.4 Characteristics of Arrays

- 1 The array name should be a valid identifier.
- 2 The name of the array should be unique, similar to other variables.
- 3 The values of the elements stored in the array should be of the same type.

### Applications of Arrays

- 1 Insertion of a value in an existing array at a particular position.
- 2 Deletion of value from an array.
- 3 Traversal of an array.
- 4 Sorting an array in to some particular order.

5 Searching for a value in an array.

6 Merging as two arrays.

### 3.5 Types of Arrays (Categories of Arrays)

There are 3 types of arrays.

- One dimensional arrays
- Two dimensional arrays
- Multi-dimensional arrays.

#### 1. One dimensional arrays

Arrays whose elements are specified by a single subscript are called as one dimensional arrays or single dimensional or single subscripted or linear arrays.

**Syntax:**

<b>Storage type</b> <b>Data type</b> <b>array name [size];</b>
--

**Accessing array elements:**

Once an array is declared then the individual elements in the array are accessed with the help of subscripts. All the array elements are numbered starting from zero. The first item is stored at the address pointed by the array name itself.

**Example:** Write a C program to find the biggest of 10 numbers using arrays.

```
#include<stdio.h>
#include<conio.h>
void main()
{
int i,n,a[10],max;
clrscr();
printf("enter n value");
scanf("%d",&n);
for(i=0;i<n;i++)
scanf("%d",&a[i]);
max=a[0];
for(i=0;i<n;i++)
if(a[i]>max)
max=a[i];
printf("maximum number=%d",max);
getch();
}
```

## 2. Double dimensional arrays

A double dimensional array is defined in the same manner as a single dimensional array except that it requires two pairs of square brackets for two subscripts.

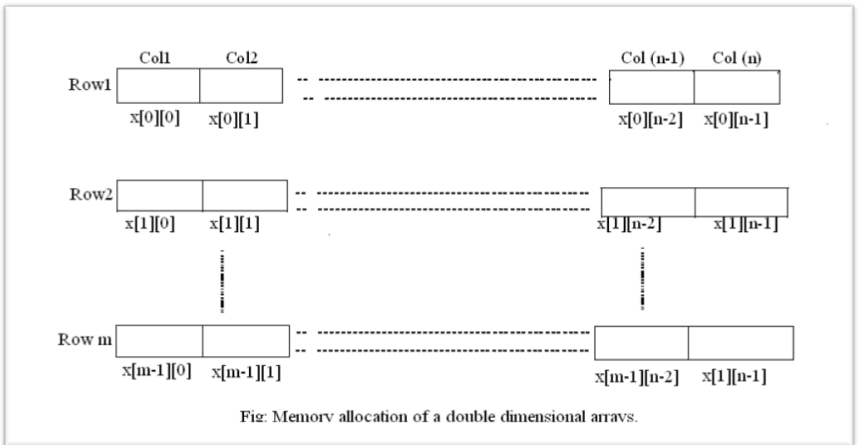
**Syntax:**

**Storage type    Data type    array name [row size ] [column size ];**

**Example:**        `int mark [2] [2];`

Defines a table as an integer array having 2 rows and 2 columns. An array element starts with an index zero and so the individual elements of an array will be

`int mark`  $\left[ \begin{array}{cc} [0][0] & [0][1] \\ [1][0] & [1][1] \end{array} \right]$



## Initializing double dimensional arrays

### Syntax:

**Storage type**      **data type**      **array name [row size][columns size]={list of values};**

### Example:

```
int mark[4][2] = {
                    { 84, 56 },
                    { 92, 67 },
                    { 75, 78 },
                    { 69, 89 }
                };
```

It can be also declared as

int mark[4][2]= { 84, 56, 92, 67, 75, 78, 69, 89 }; The result of the initial assignment are

Mark[0][0]=84	mark[1][0]=56
Mark[0][1]=92	mark[1][1]=67
Mark[0][2]=75	mark[1][2]=78
Mark[0][3]=69	mark[1][3]=89

**Example:** Write a C program to calculate the sum of all elements in a matrix using double dimensional array.

```
#include<stdio.h>
#include<conio.h>
void main()
{
int a[10][10], i,j,m,n,sum=0;
printf("Enter the order of matrix");
scanf("%d%d",&m,&n);
printf("Enter the elements of a matrix");
for(i=0;i<m;i++)
{
for(j=0;j<n;j++)
{
scanf("%d",&a[i][j]);
sum=sum+a[i][j];
}
}
}
```

```

}
printf("Sum of the matrix are %d",sum);
getch();
}

```

### 3. Three dimensional or multi-dimensional array

The C program allows array of two or multi dimensions.

Syntax: Data type array name [s1][s2][s3].....[sn]

Initialization:

```

int mat[3][3][3]= {
                    { 1, 2, 3,
                      4, 5, 6,
                      7, 8, 9,
                      1, 4, 7,
                      2, 8, 9,
                      1, 2, 3
                    }
};
{
                    2, 9, 8,
                    4, 1, 3,
                    3, 2, 3
}

```



A three dimensional array can be thought of as an array of arrays. The outer array contains three elements the inner array size is two dimensional with size [3] [3]

**Example:** Write to C program to explain the working of three dimensional arrays

```
#include<stdio.h>
#include<conio.h>
void main()
{
int a3d[3][3][3];
int a,b,c;
clrscr();
for(a=0;a<3;a++)
for(b=0;b<3;b++)
for(c=0;c<3;c++)
a3d[a][b][c]=a+b+c;

for(a=0;a<3;a++)
{
printf("\n");
for(b=0;b<3;b++)
{
```

```
    for(c=0;c<3;c++)
        printf("%3d", a3d[a][b][c]);
        printf("\n");
    }
}
getch();
}
```

### **Limitations of arrays**

1. The compiler uses static memory allocation for an array that is it is not possible to increase or decrease the array size at runtime.
2. Elements cannot be inserted into an array.
3. We cannot delete elements into an array.
4. If the number of elements to be stored is not known in advance, there may be memory waste if an array of large size is specified.
5. If a small array size is specified there may not be enough memory to place all elements.
6. C does not perform bound checking of an array.

## Difference between character array and integer array

Character array	Integer array
Character array NULL (\0) character is automatically added at the end.	Integer array or other types of arrays no NULL character is placed at the end.

### 3.6 STRINGS

1. A group of characters can be stored in a character array. These character arrays are called strings.
2. To recognize a character array should end with a NULL character ('\0').

**Example:** The string “student” could be stored as

's'	't'	'u'	'd'	'e'	'n'	't'	'\0'
-----	-----	-----	-----	-----	-----	-----	------

3. The length of a string is the number of characters it contains excluding NULL character. But, to store a string, we need one more locations than the length of the string.

**3.7 Declaring strings:-**The general form of declaration of a string variable is

**Syntax:** char variable-name [size];

**Example:** char name [10];

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

The scanf( ) function usually accepts an address of the variable but in arrays ,by itself indicates the starting address of the array and hence we don't use an ampersand (&) before the variable name while reading strings.

### 3.8 Initializing strings

**Syntax:** char name [size] = { list of characters };

**Example:** char name [25] = { 'r' 'a' 'v' 'i' , '\0' };

```
char name [ ] = { 'r' 'a' 'v' 'i' , '\0' };
```

```
char name [25] = "ravi";
```

```
char name [ ] = "ravi";
```

Note: While initializing a character array by listing its elements, must specify the NULL terminator ('\0') with arrays elements.

**Example:** Write a C program to display the output when the count of NULL character is not considered.

```
#include<stdio.h>
#include<conio.h>
void main()
{
char name[6]={'w','e','l','c','o','m'};
printf("name=%s", name);
getch();
}
```

**Output:** welcome followed by garbage characters

**Explanation:** The output of the above program would be welcome followed by some garbage values. To get the correct result the argument must be [7] instead of [6]. The output can be seen as given below after changing the argument [7] in place of [6].

**Example:** Write a C program to print “welcome” by using different formats of initialization of array.

```
#include<stdio.h>
#include<conio.h>
void main()
{
```

```

char a1[9]={'w','e','l','c','o','m','e','\0'};
char a2[9]="welcome";
char a3[9]={ {'w'},{'e'},{'l'},{'c'},{'o'},{'m'},{'e'} };
clrscr();
printf("\n a1=%s",a1);
printf("\n a2=%s",a2);
printf("\n a3=%s",a3);
getch();
}

```

**Example:** Write a C program to display the string “prabhakar” using various format specifications.

```

#include<stdio.h>
#include<conio.h>
void main()
{
char a1[15]="prabhakar";
clrscr();
printf("\n %s",a1);
printf("\n %.5s",a1);
printf("\n %.8s",a1);
printf("\n %.15s",a1);
printf("\n %-10.4s",a1);

```

```
printf("\n %11s",a1);  
getch();  
}
```

Output:

Prabhakar

Prabh

Prabhaka

prabhakar

prab

Prabhakar

**3.9 Reading the strings:** There are three ways to read a string from the user through the keyboard. They are

1. By using scanf ( ) function
2. By using gets ( ) function
3. By using loops

**1. By using scanf ( ) function:** The scanf( ) function with format string %s can be used to read a string from the user.

**Example:**

```
void main()  
{  
char name[10];  
scanf("%s", name);  
getch();  
}
```

```
}
```

The advantage of using scanf() function is that it can be used to read more than one string at a time.

Example: scanf(“%s %s %s”, name ,designation, address);

The disadvantages of scanf ( ) is, it terminate when it encounters a blank space.

**2. By using gets ( ) function:** The gets ( ) function overcomes the disadvantages of scanf( ) function, since it can read a string of any length with any number of blank spaces and tabs . It gets terminated only when an “ENTER” key is pressed.

**Example:**

```
void main()
{
char name[10];
gets(name);
getch();
}
```

The disadvantages of gets( ) function is that it can be used to read only a single string at a time.



**3.By using the loops:** A string is an array of characters, hence a string can also be read, character-by-character from the user by using loops.

```
void main()
{
char name[10];
for(i=0;i!='\0';i++)
scanf("%c", &name[i]);
getch();
}
```

Note: Ampersand (&) is used before the array name if we are reading a string character-by-character.

**3.10 Writing strings:** The puts( ) function writes the character string is supplied as a parameter to the standard output device.

**Example:** puts(name);

### 3.11 String library functions:

Various library functions in string.h

Function	Purpose	Example	Result
Strupr ( )	To convert all alphabets in string to upper case letters	Strupr(“srist”)	SRIST
Strlwr()	To convert all alphabets in string to lower case letters.	Strlwr(“SRIST”)	Srist
Strlen()	Finds the length of the string in bytes excluding NULL character	Char s[]=”city”; Int n; n=strlen(s);	4
Strrev()	To reverse a string	Strrev(“city”);	Ytic
Strcpy()	Copies string str2 to string str1	Char s2[]=”city” Char s1[20]; Strcpy(s1,s2);	S2 contains city

Strcmp()	Used to compare if two strings are identical	n=strcmp("srist", "srist")	n=0
Strncmp()	Compares the first n characters of s1 and s2	n=strncmp("raj", "ram", 2);	n=0
Strcat()	To join s2 to s1	Strcat("stu", "dent")	student

**Example 1:** Write a C program to find the length of a given string

```
#include<strig.h>
```

```
void main()
```

```
{
```

```
char str[10];
```

```
printf("Enter the string");
```

```
gets(str);
```

```
printf("The length of a given string is %d", strlen(str));
```

```
getch();
```

```
}
```

**Example 2:** Write a C program to copy from one string into another string

```
void main()
{
char s1[10];
char s2[20];
printf("Enter the string");
gets(s1);
printf("The copied string is %s",strcpy(s2,s1));
getch();
}
```

**Example3:** Write a C program to perform string concatenation

```
void main()
{
char s1[10];
char s2[20];
printf("Enter the string 1");
gets(s1);
printf("Enter the string 2");
gets(s2);
printf("The copied string is %s", strcat(s1,s2));
```

```
getch();  
}
```

**Example 4:** Write a C program to reverse the given string

```
void main()  
{  
char s1[10];  
printf("Enter the string 1");  
gets(s1);  
printf("The reverse string is %s", strrev(s1));  
getch();  
}
```

**Example 5:** Write a C program to convert from upper to lower case

```
void main()  
{  
char s1[10];  
printf("Enter the string 1");  
gets(s1);  
printf("The lower case string is %s", strlwr(s1));  
getch();  
}
```

**Example 6:** Write a C program to convert from lower to upper case

```
void main()
{
char s1[10];
printf("Enter the string 1");
gets(s1);
printf("The upper case string is %s",strupr(s1));
getch();
}
```

**Example 7:** Write a C program to check the strings are equal or not

```
void main()
{
char s1[10];
char s2[10];
printf("Enter the string 1");
gets(s1);
printf("Enter the string 2");
gets(s2);
x=strcmp(s1,s2);
if(x==0)
```

```
printf("strings are equal");
else
printf("strings are not equal");
getch();
}
```

**Example:** Write a C program to perform the string operations using string library functions.

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
void main()
{
char s[20],s2[20];
int x,ch;
clrscr();
printf("Enter the two strings");
scanf("%s %s",s,s1);
do
{
printf("\n 1.strlen");
printf("\n 2.strupr");
printf("\n 3.strlwr");
```

```
printf("\n 4.strrev");
printf("\n 5.strcat");
printf("\n 6.strcpy");
printf("\n 7.strcmp");
printf("\n 8.exit");
printf("\n Enter the choice");
scanf("%d",&ch);
switch(ch)
{
    case 1:printf("The length of the string is %d",strlen(s));
           break;
    case 2:printf("The upper case of the string is %s",strupr(s));
           break;
    case 3:printf("The lower case of the string is %s",strlwr(s));
           break;
    case 4:printf("The reverse of the string is %s",strrev(s));
           break;
    case 5:printf("The string concatenation is %s",strcat(s,s1));
           break;
    case 6:printf("The copied string is %s",strcpy(s,s1));
           break;
    case 7:if(x==strcmo(s,s1))
```



```
printf("The strings are equal);
else
printf("The strings are not equal");
    break;
case 8:exit(0);
default: printf("Invalid choice");
}
printf("Do u want continue press(y/n) buttons");
ch=getch();
}
while(ch=='y');
getch();
}
```

**Example:** Write a C program to find the length of a given string by using user defined functions

```
#include<stdio.h>
#include<conio.h>
void main()
{
char str[10];
int i;
```

```
printf("Enter the string");
gets(str);
for(i=0;str[i]!='\0'; i++);
printf("The length of a string is %d", i);
getch();
}
```

**Example:** Write a C program to perform string copy by using user defined functions

```
#include<stdio.h>
#include<conio.h>
void main()
{
char str[10],str1[10];
int i;
printf("Enter the string");
gets(str);
for(i=0;str[i]!='\0'; i++)
{
str1[i]=str[i];
}
str1[i]='\0';
printf("The copied string is %s",str1);
```

```
getch();  
}
```

**Example:** Write a C program to perform string concatenation by using user defined functions

```
#include<stdio.h>  
#include<conio.h>  
void main()  
{  
char str[10],str1[10];  
int I, j;  
printf("Enter the string 1");  
gets(str);  
printf("Enter the string 2");  
gets(str1);  
for(i=0;str[i]!='\0'; i++);  
for(j=0;str1[j]!='\0';j++,i++)  
{  
str[i]=str1[j];  
}  
str[i]='\0';  
printf("The string concatenation is %s", str);  
getch();
```

```
}
```

**Example:** Write a C program to reverse the given string by using user defined functions

```
#include<stdio.h>
#include<conio.h>
void main()
{
char str1[10],rev[10];
int I, j ,n=0;
printf("Enter the string");
gets(str1);
for(i=0;str[i]!='\0'; i++)
n++;
for(j=n-1,i=0;j>=0;j--,i++)
{
rev[i]=str1[j];
}
rev[i]='\0';
printf("The reverse string is%s", rev);
getch();
}
```

**Example:** Write a C program to check whether the strings are equal or not by using user defined functions.

```
#include<stdio.h>
#include<conio.h>
void main()
{
char str1[10],str2[10];
int i,j,flag;
printf("Enter the string1");
gets(str1);
printf("Enter the string2");
gets(str2);
for(i=0;str1[i]!='\0'; i++)
{
for(j=0;str2[j]!='\0';j++)
{
if((str1[i]!=str2[j])&&(i==j))
{
flag=1;
}
}
}
}
```

```
if((flag==1)||(i!=j))
printf("The strings are not equal");
else
printf("The strings are equal");
getch();
}
```

**Example:** Write a C program to check whether the string is palindrome or not.

```
#include<stdio.h>
#include<conio.h>
void main()
{
char str[10];
int i, j,flag=0, n;
printf("Enter the string");
gets(str);
n=strlen(str);
for(i=0, j=n-1; str[i]!='\0'; i++, j--)
{
if(str[i]!=str[j])
{
flag=1;

```

```
break;
}
}
if((flag==0)
printf("The string is Palindrome");
else
printf("The string is not palindrome");
getch();
}
```

## UNIT-IV

### **Pointers & User Defined Data types**

Pointers, dereferencing and address operators, pointer and address arithmetic, array manipulation using pointers, **User-defined data types**-Structures and Unions.

**4.1 Definition:** A pointer is memory variable that stores a memory address. Pointer can have any name that is legal for other variable and it is declared in the same fashion like other variable but it is always denoted by “\* “operator.

#### **4.2 Advantages of pointers**

1. Pointers save the memory space.
2. Pointers are used to increase the speed of execution.
3. Pointers are used to reduce the length and complexity of a program.
4. Pointers are useful for representing two dimensional and multi dimensional arrays.
5. The memory is accessed efficiently with the pointers.

#### **4.3 Operators used with pointers**

**a)Indirection operator or dereference operator (\*):** It is used in two distinct ways with pointers



1. To declare a pointer variable in the declaration section, before the identifier \* is used. Then that variable is converted into pointer variable.
2. To access the value stored in a particular memory location \* operator is used. That is why it is also called dereference operator.

**b)Address operator (&):** It is used to find the address of a variable. If we put & before any variable name, we get the address of that memory location.

**4.4 Pointer declaration:** Like all variables, a pointer variable should also be declared.

Syntax: Data type \* variable name;

Example: int \*ptr;



**Fig: Representation of a pointer variable**

**Example: Write a c program for accessing a variable through a pointer**

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

```
{  
int a=5,b=10,*ptr;  
ptr=&a;  
printf("before values a=%d b=%d",a,b);  
b=*ptr;  
printf("after values a=%d b=%d",a,b);  
printf("value of ptr%d",ptr);  
printf("address of a is %d", &a);  
printf("value of *ptr is%d",*ptr);  
getch();  
}
```

Output: Before values a=5 b=10

After values a=5 b=5

Value of ptr is 4040

Address of a is 4040

Value of \*ptr is 5

**4.5 Initialization of pointers:** A pointer is similar to any other variable except that it holds only memory address, and therefore, it needs to be initialized with a valid address before it can be used.

Pointers should be initialized either when they are declared or in an assignment statement. A pointer may be initialized to NULL or with the address of some other variable, which is already defined.

Example: `ptr = &i` is called as pointer initialization.

**4.6 Pointer arithmetic:** Arithmetic operations on pointer variables are also possible. ++, --, prefix and postfix operations can be performed with the help of the pointers. The effect of these operations are shown in the below given table.

Data type	Initial address	Operation	Address	after operations	Required bytes
<code>int i=2</code>	4046	++	-- 4048	4044	2
<code>char c='x'</code>	4053	++	-- 4054	4052	1
<code>float f=2.2</code>	4058	++	-- 4062	4054	4
<code>long l=2</code>	4060	++	-- 4064	4056	4

**Note:** ++ to be incremented, but not by 1. -- to be decremented, not by 1.

**Example: write a c program to display the address of the variable**

```
#include<stdio.h>
#include<conio.h>
void main()
{
int n;
printf("Enter the n value");
scanf("%d",&n);
printf("value of num=%d", num);
printf("address of num=%d",&num);
getch();
}
```

Output: n=20

value of num=20

Address of num=4066

**Example: Write a c program to find the sum of two values using pointers**

```
#include<stdio.h>
#include<conio.h>
void main()
{
int a, b, sum, *ptr1,*ptr2;
ptr1=&a;
ptr2=&b;
printf("enter the two numbers");
scanf("%d %d",&a,&b);
sum=*ptr1 +*ptr2;
printf("sum=%d", sum);
getch();
}
```

**Example: Write a c program to find the biggest of two values using pointers**

```
#include<stdio.h>
#include<conio.h>
void main()
{
int a, b, *big *ptr1,*ptr2;
```

```

ptr1=&a;
ptr2=&b;
printf("enter the two numbers");
scanf("%d %d",&a,&b);
if(*ptr1>*ptr2)
big=ptr1;
else
big=ptr2;
printf("biggest number is=%d", *big);
getch();
}

```

**4.7 Pointers and arrays:** In C, when an array is declared, the array name denotes the address of the 0th element in that array. The address of the 0th element is also called base address or starting address. Therefore, if we refer the name of the array we get the starting address of the array.

So array name itself is an address or pointer. The elements of the array together with their address can be displayed by using array name itself since array elements are stored in contiguous memory locations.

For example, an array is initialized as bellow

```
int a[4]= {3,7,9,4};
```

Now, we can access the value in the second location by two ways.

By array notation: `a[1]`

By pointer notation: `*(a+1)`

Similarly, we can get address of the second location by two ways.

By array notation: `&a[1]`

By pointer notation: `(a+1)`

**Example: Write a C program to find the biggest number among given list of elements by using pointers**

```
#include<stdio.h>
#include<conio.h>
void main()
{
int i,n,a[20], *ptr;
clrscr();
printf("enter n value");
scanf("%d",&n);
for(i=0;i<n;i++)
scanf("%d",&a[i]);
*ptr=a[0];
for(i=0;i<n;i++)
```

```

if(a[i]>*ptr)
*ptr=a[i];
printf("maximum number=%d",*ptr);
getch();
}

```

**Example: Write a C program to display element name, value at that location and address of that location.**

```

#include<stdio.h>
void main()
{
int x[]={1,2,3,4};
int i;
printf("element.no, elem address\n");
for(i=0;i<4;i++)
{
printf("x[%d] %d %d", i, *(a+i), (a+i));
}
getch();
}

```

**Output:**

Element. No	element	Address
X[0]	1	4056



X[1]	2	4058
X[2]	3	4060
X[3]	4	4062

**4.8 Dynamic memory allocation:** Memory allocations mean allocating sufficient memory space to all the variables in the program. Memory space can be allocated in two ways. They are

1. Static memory allocation
2. Dynamic memory allocation

**1. Static memory allocation:** To allocating the memory at the time of compilation, then it is said to be a static memory allocation.

**2. Dynamic memory allocation:** To allocating the memory at the time of execution, then it is said to be dynamic memory allocation.

In C language there are 4 ways to allocating the memory

- 1.The malloc( ) function
- 2.the calloc( ) function
- 3.The realloc( ) function
- 4.The free( ) function

**1. The malloc ( ) function:** This is used to allocate a contiguous block of memory in bytes.

Syntax:

Pointer variable=(cast-type \*) malloc(size);

Example: x=(int \*)malloc(20);

x=(int \*)malloc(10 \*sizeof(int)); on execution of malloc, 10 times the size of an int (ie.,  $10*2=20$  bytes) is allocated and the starting address of the first byte is assigned to pointer x of type int.

Where pointer-variable is a valid C variable already defined. Cast -type is the type of the pointer returned by malloc ( ) such as int, char...etc. size is the required size of memory in byte.

**Note:** If the memory allocation is success it returns the starting address else it returns the NULL.

**Example: Write a C program to perform sum of array elements by using malloc( ) function**

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

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

```
void main()
```

```
{
```

```
int *a,i,n,sum=0;
```

```
printf("enter the size of array");
scanf("%d",&n);
a=(int *)malloc(n*size(int));
if(a!=NULL)
{
    printf("enter the elements");
    for(i=0;i<n;i++)
    {
        scanf("%d",&a[i]);
        sum=sum+a[i];
    }
    printf("sum of given elements are %d",sum);
}
else
printf("memory can not be allocated");
getch();
}
```

**Output:** Enter the size of the array 5  
Enter the elements 1 2 3 4 5  
Sum of given elements are 15

**2. The calloc ( ) function:** This function is used to allocate multiple blocks of contiguous memory in bytes. All the blocks are of same size.

Syntax: pointer-variable =(cast-type \*) calloc(n, size);

Example: x=(int \*) calloc(5,10);

Where pointer-variable is a valid C variable already defined. Cast-type is the type of the pointer returned by calloc ( ) such as int, char ....etc. 'n' is the number of blocks and size is the required size of memory in bytes.

Note: If the memory allocation is success it returns the starting address else it returns NULL.

**3. The realloc ( ) function:** The realloc ( ) function is used to increase or decrease the size of memory previously allocated by using malloc/calloc ( ) function.

Syntax: new-pointer=realloc (old-pointer, new-size);

Where new-pointer is valid C variable previously defined. Old-pointer is the pointer variable used in malloc( ) or calloc( ) function. New-size is the size of the new memory needed.

**Example:** y= (int \*) malloc(50);

X=realloc(y, 30);

The first statement allocated memory space of size 50 bytes and returns the starting address of the memory through the pointer x. the second statement reallocates (decreases) the already allocated space to 30 bytes.

**4. The free ( ) function:** The free ( ) function is used to free (release or de allocate) the block of unused or already used memory.

Syntax:            free (pointer-variable);

Where pointer-variable is a pointer to the memory block which has been already created by malloc ( ) or calloc ( ) function.

**Example:** x= (int \*) malloc (50);  
                  free (x);

The first statement allocated memory space of 50 bytes and returns the starting address of the allocated memory through a pointer variable x. The second statement frees the allocated memory.

**4.9 Definition:** A structures is a collection of one or more variables of different types, grouped together under a single name.

The variables or data items in a structure are called as members of the structures.

**4.10 Declaration of structures:** The general form or the syntax of declaring a structure is

```
struct <structure name>
{
data type member1;
data type member2;
.
.
.
data type member N;
};
```

In the above declaration, struct is a keyword followed by an optional user defined structure name usually referred as a tag. The list of member declaration is enclosed in a pair of flower braces. The closing brace of the structure and the semicolon ends the structure declaration.

**Example:**

```
struct employee
{
int empno;
char  ename[10];
float salary ;
};
```

Where employee is the name of the structure. The structure employee contains three members of type int, char and float representing empno, empname and salary respectively.

**4.11 Defining a structure:** Defining a structure means creating variables to access members in the structures. Creating structure variables allocates sufficient memory space to hold all the members of the structures.

The syntax for defining the structure during structure its declaration is

```
struct <structure name>
{
data type member1;
data type member2;
.
```

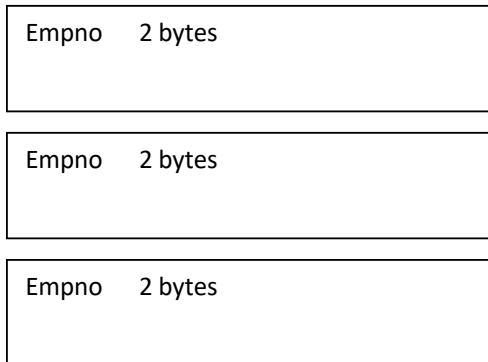
.  
. .

```
data type member N;  
}structure-variable(s);
```

**Example:**

```
struct employee  
{  
int empno;  
char ename[10];  
float salary ;  
} emp1;
```

Fig: Shows the memory allocation for the structure employee



**Fig: Memory occupied by structure employee**



**4.12 Accessing structure members:** Dot (.) or period operator is used to access a data member of a structure variable. It is to be noted that the dot operator separates the structure variable and the data member.

**Syntax:** structure-variable . Member-name

**Example: Write a C program to declaring, initializing and accessing members of structure by dot operator**

```
void main()
{
struct book
{
char title[40];
int pages;
float price;
};
struct book b1= { "pc&ds", 300,150.0};
printf("%s\n", b1.name);
printf("%d\n", b1.pages);
printf("%f\n", b1.price);
getch();
}
```

**4.13 Initialization of structures:** The structure members can be initialized only by using structure variables during structure declarations.

**Syntax:**

```
struct <structure name>
{
data type member1;
data type member2;
..
.
.
data type member N;
}structure-variable={ list of values};
```

**Example:**

```
struct employee
{
int height;
float weight ;
} emp1={154, 77.9};
```

The above initialization initialize 154 to the structure member height and 77.9 to the structure member weight.

The values to be initialized for the structure members must be enclosed with in a pair of braces.

**Example: Write a C program to initializing the structure using the structure name**

```
#include<stdio.h>
#include<conio.h>
struct student
{
int rollno;
char name[10];
}s3;
void main()
{
struct student s1= {123, "sai"};
struct student s2= {456, "kumar"};
printf("name is %s\n", s1.name);
printf("rollno is %d\n", s1.rollno);
printf("name is %s\n", s2.name);
printf("rollno is %d\n", s2.rollno);
s3=s1;
printf("name is %s\n", s3.name);
printf("rollno is %d\n", s3.rollno);
```

```
getch();  
}
```

**Example: Write a C program to assign the values to members**

```
#include<stdio.h>  
#include<conio.h>  
void main()  
{  
struct student  
{  
int rollno;  
char name[10];  
char branch[20];  
};  
struct student s  
strcpy(s.name, "sai");  
strcpy(s.branch, "cse");  
s.rollno=32;  
printf("%s %s %d\n", s.name, s.branch, s.rollno);  
getch();  
}
```

**Example: Write a C program to assign the values from the keyboard**

```
#include<stdio.h>
#include<conio.h>
void main()
{
struct account
{
int number;
char name[10];
float balance;
}a;
printf("Enter the account holder name");
gets(a.name);
printf("Enter the account number");
scanf("%d",&a.number);
printf("Enter the balance");
scanf("%f",&a.balance);
printf("The details are ");
printf("%s %d %f\n", a.name, a.number, a.balance);
getch();
}
```

**4.14 Nested structures:** Nested structures are nothing but a structure with in another structure. A structure may be defined and/or declared inside another structures.

Example:

```
struct employee
{
int empno;
char name[10];
struct employ_add
{
int no;
char street[10];
char area[10];
long int pincode;
}address;

char deptname[10];
float salary;
}emp1, emp2;
```

In the above structure declaration employee is the main structure. It gets additional information about the

employ\_add. The member of a nested structure is accessed is

Main structure variable. Sub structure variable . Sub structure member

**Example:** emp1. address. no

**Example: Write a C program on nested structures**

```
#include<stdio.h>
#include<conio.h>
void main()
{
    struct employee
    {
        int empno;
        char name[10];
        struct employ_add
        {
            int no;
            char street[10];
            char area[10];
            long int pincode;
        }address;
        char deptname[10];
        float salary;
```

```
}emp1;
printf("enter the empno, empname, deptname, salary of the
employee");
scanf("%d %s %s %f",&emp1.empno, emp1.empname,
emp1.deptname,&emp1.salary);
printf("Enter the address of the employee");
scanf("%d %s %s %s %ld", &emp1.address.no,
emp1.address.street, emp1.address.area,
emp1.address.pincod);
printf("\n employee no=%d",emp1.empno);
printf("\n employee name=%s",emp1.empname);
printf("\n department name=%s",emp1.deptname);
printf("\n salary=%f",emp1.salary);

printf("employee address no=%d", emp1.address.no);
printf("employee street=%s", emp1.address.street);
printf("employee pincod =%ld", emp1.address.pincod);
getch();
}
```



**4.15 Array of structures:** It is possible to declare array of structures. The following example illustrates this.

```
Struct employee_info
{
    char name[30];
    int age;
    char designation[10];
    float salary;
}staff [200];
```

This declares staff to be an array with 200 elements. The process of declaring a structure array is similar to declaring any other kind of array.

**Example: Write a C program to declaring array of structure student and displaying the marks of students who got more than 75 marks.**

```
#include<stdio.h>
#include<conio.h>
Struct student
{
    int rno;
    char sname[10];
    int marks;
```

```

};
void main()
{
    struct student s[60];
    int i, n;
    printf("How many students");
    scanf("%d",&n);
    for(i=0;i<n;i++)
    {
        printf("Enter the rollno");
        scanf("%d",&s[i].rno);
        printf("Enter the name");
        scanf("%s", s[i].name);
        printf("Enter the marks");
        scanf("%d",&s[i].marks);
    }
    printf("Rollno  name  marks");
    for(i=0;i<n;i++)
    {
        if(s[i].marks>=75)
            printf(" %d  %s  %d", s[i].rno, s[i].sname, s[i].marks);
    }
}

```

```
getch();  
}
```

Array of structures can be initialized as  
struct employee

```
{  
int empno;  
char empname[20], deptname[20];  
float salry;  
}emp[5]={ {1, "kumar" , "sales", 300.50},  
          {2, "subbu" , "accounting", 400.50},  
          {3, "manoj" , "marketing", 583.50},  
          {4, "madhu" , "production", 88750.50},  
          {5, "sudha" , "maintaintence", 7235.50}  
};
```

**Note:** If some of the members of the structures are not initialized it takes a value zero. If the member is a char data types , it takes a value NULL.

**Example: Write a C program to initialize the array of structure**

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

```
{
int i;
clrscr();
struct employee
{
int empno;
char empname[20];
float salry;
}emp[3]={ {1, "kumar", 300.50},
          {2, "subbu", 400.50},

          };
for(i=0;i<3;i++)
{
printf("employee name=%s", emp[i].empname);
printf("employee no=%d", emp[i].empno);
printf("employee salary=%f",emp[i].salary);
}
getch();
}
```

**4.16 Pointers and structures:** Members of a structure can be accessed by a pointer. To access the structure members by pointer we use → (arrow) operator.

Example:

```
#include<stdio.h>
#include<conio.h>
struct book
{
    char title[10];
    int pages;
    float price;
};
void main()
{
    struct book *ptr;
    struct book b1;
    ptr=&b1;
    strcpy(ptr->title, "pc&ds");
    ptr->pages=300;
    ptr->price=150.0;
    printf("title=%s",b1.title);
    printf("pages=%d", b1.pages);
```

```
printf("price=%f", b1.price);  
getch();  
}
```

**4.17 Functions and structures:** There are three ways to pass a structure to a function

1. Passing structure members to functions
2. Passing the address of members to functions
3. Passing entire structure to function.

**1. Passing structure members to functions:** This method is used to pass members of the structure as actual arguments of the function call statement.

Example:

```
struct employee  
{  
int empno;  
char empname[20];  
char deptname[20];  
float salary;  
}emp1;
```

The members of the structure can be passed to the function employ ( ) as  
employ (emp1.empno);

```
employ (emp1.empname);  
employ (emp1.salary);  
employ(emp1.deptname);
```

This method is the most common method and becomes inefficient when the structure is large.

Example: Write a C program to pass the structures members to functions

```
#include<stdio.h>  
#include<conio.h>  
struct employee  
{  
int empno;  
char empname[20];  
}emp1;  
  
void employ();  
void main()  
{  
printf("\n enter the employee no and name");  
scanf("%d %s", &emp1.empno, emp1.empname);  
employ(emp1.empno,emp1.empname);  
getch();
```

```
}  
void employ()  
{  
printf("\n the employee no is %d", emp1.empno);  
printf("\n the employee name is %s", emp1.empname);  
}
```

**2. Passing the address of members to functions:** Members of a structure can also be passed to a function by passing their address. In this method, the address location of the members is passed to the called function, hence the address operator (&) is used before the structure name. The members of the structure can be passed to a function employ () as

```
employ (&emp1.empno);  
employ (&emp1.salary);  
employ (emp1.deptname);  
employ (emp1.empname);
```

This method is more efficient than the previous one, since it, works faster, but becomes inefficient when the structure is large.



Example: Write a C program to pass the address of members to function

```
#include<stdio.h>
#include<conio.h>
struct employee
{
int empno;
char empname[20];
}emp1;
void employ();
void main()
{
printf("\n enter the employee no and name");
scanf("%d %s", &emp1.empno, emp1.empname);
employ(&emp1.empno,emp1.empname);
getch();
}
void employ()
{
printf("\n the employee no is %d", emp1.empno);
printf("\n the employee name is %s", emp1.empname);
}
```

**3. Passing entire structure to function:** In this method, the entire structure is passed as an argument to the function.

Example: Write a C program to pass the entire structure to functions

```
#include<stdio.h>
#include<conio.h>
struct employee
{
int empno;
char empname[20];
}emp1;
void employ(struct employee emp);
void main()
{
printf("\n enter the employee no and name");
scanf("%d %s", &emp1.empno, emp1.empname);
employ(emp1);
}
void employ(struct employee emp)
{
printf("\n the employee no is %d", emp.empno);
printf("\n the employee name is %s", emp.empname);
```

```
getch();  
}
```

**4.18 Typedef:** The typedef statement defines synonyms for an existing data type. It does not introduce a new data type and does not reserve storage also.

**Syntax:**

```
typedef existing data type new data type
```

**Example:**

```
typedef float REAL;  
REAL area, volume;
```

**Example: Write a C program to create user defined data type hours on int data type and use it in the program.**

```
#define H 60  
  
void main()  
{  
typedef int hours;  
hours hrs;  
clrscr();  
printf("enter the hours");  
scanf("%d", &hrs);  
printf("\n minutes=%d", hrs*H);
```

```
printf("\n seconds=%d", hrs*H*H);  
getch();  
}
```

**4.19 Unions:** A union is a derived data type, like structure, but only one data member is active at a time. In structure each member has its own memory location whereas, members of unions have same memory locations. Unions also contains members of types int, float , char , long, arrays, pointers.....etc.

Syntax:

```
union <union name>  
{  
data type member1;  
data type member2;  
.  
.  
.  
data type memberN;  
}union variable;
```

The syntax of union is identical to that of a structure except that the keyword struct is replaced with the keyword union.

Example:

```
union exam
{
int rollno;
char name[10];
int m1,m2,m3;
};
```

In the above example, union has 5 members. First member is a character array name having 10 characters (i.e., 10 bytes). Second member is of type int that requires 2 bytes for their storage. All the other members m1, m2, m3 are integers which requires 2 bytes for their storage.

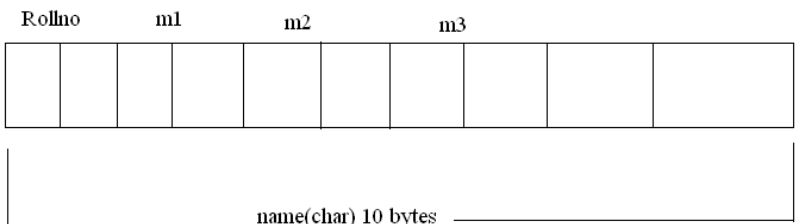


Fig:memory occupied by union exam

The fig: shows the memory allocation of members of union exam

In union, all these 5 members are allocated in a common place of memory.

**Example: Write a C program to show how many bytes are occupied by structure and union.**

```
union exam
{
int rollno;
char name[10];
int m1,m2,m3;
}u1;
struct exam1
{
int rollno;
char name[10];
int m1,m2,m3;
}s1;
void main()
{
printf("The size of union is %d\n", sizeof(u1));
printf("The size of structure is %d\n", sizeof(s1));
getch();
}
```

**Output:** The size of union is 10

The size of structure is 18

**4.20 Initialization a union:** There is a major difference between structure and union in terms of storage. In structure, each member has its own storage location where as all the members of union occupy the same memory location.

A union may contain many members of different data types; it can handle only one member at a time. Hence, we can initialize only one member in a union.

If we try to initialize more than one member, the last initialized value will be assigned to all of its members.

**Example: Write a C program to demonstrate initialization of a union**

union exam

```
{  
int rollno,m1,m2,m3;  
}u1;  
void main()
```

```
{
u1.rollno=252;
u1.m1=89;
printf("\n Roll no=%d\n", u1.rollno);
printf("\n m1=%d\n", u1.m1);
printf("\n m2=%d\n", u1.m2);
printf("\n m3=%d\n", u1.m3);
getch();
}
```

Output:

Rollno=89

M1=89

M2=89

M3=89

**4.21 Self-referential structures:** In a structure, if one or more members are pointers pointing to the same structure, then that structure is called self-referential structure. In simple, a structure refers to itself is known itself referential structure.

**Example:**

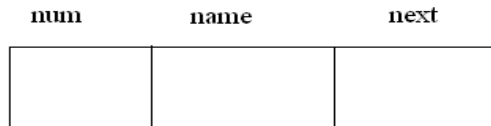
Struct node

```
{
```



```
int num;
char name[10];
struct node *next;
};
```

Each node consists of three data items 1.number 2.name 3.next node. The pointer variable next is called a link. These structures are represented as follows:



The pointer “next” contains either an address of the location in memory of the successor node element or NULL. The NULL is used to denote the end of the list. We now declare three nodes as structure type node:

```
node n1, n2, n3;
```

We assign data values to these nodes:

```
n1.num=10; strcpy(n1.name, "ravi");
n2.num=20; strcpy(n2.name, "kumar");
n3.num=30; strcpy(n3.name, "krishna");
```

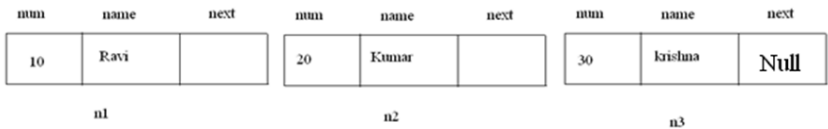
Let us link these nodes together

```
n1.next=&n2;
```

```
n2.next=&n3;
```

```
n3.next=NULL;
```

These pointers assignments result in linking n1 to n2 to n3.



**Fig: A linked list**

Now the links allow us to retrieve data from successive nodes. Thus

`n1.next->num` and `n1.next->name` have values 20 and "kumar"

`n1.next->next->num` and `n1.next->next->name` have values 30 and "krishna"

Example: Write a C program on self-referential structures.

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

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

```
void main()
```

```
{
```

```
struct node
```

```
{
```

```
int num;
```

```

char name[10];
struct node *next;
};
struct node n1,n2, n3;
n1.num=10; strcpy(n1.name, "ravi");
n2.num=20; strcpy(n2.name, "kumar");
n3.num=30; strcpy(n3.name, "krishna");
n1.next=&n2;
n2.next=&n3;
n3.next=NULL;
printf("\n %d %s", n1.next->num,n1.next->name);
printf("\n  %d  %s",n1.next->next->num,n1.next->next-
>name);
getch();
}

```

**Output:** 20 kumar  
30 Krishna

#### 4.22 Difference between structure and unions

Structure	Union
1. The key word struct is used to declare a structure.	1. The key word union is used to declare an union.

2. All data members in a structure are active at a time.	2. Only one data member is active at a time.
3. All the members of a structure can be initialized.	3. Only the first of union can be initialized.
4. Each members in a structure occupies and use its own memory space.	4. All union members use the same memory space.
5. More memory space is required, since each member is stored in a separate memory locations.	5. Less memory space is required since all members are stored in the same memory locations.
<p>Example:</p> <pre>Struct book {   Char title[40];   Int pages;   Float price; } s;</pre> <p>For s total memory required is 40+2+4 bytes.</p>	<p>Example:</p> <pre>Union book {   Char title[40];   Int pages;   Float price; } s;</pre> <p>For s total memory required is 40 bytes only.</p>

### 4.23 Difference between arrays and structure

<b>Arrays</b>	<b>Structures</b>
1.An array is a collection of data items of same data types.	1.A structure is a collection of data items of different data types.
2.The individual entries in an array are called elements.	2.The individual entries in a structure are called members.
3. An array declaration reserves enough memory space for its elements.	3.The structure definition reserves enough memory space for its members.
4.There is no keyword to represent arrays but the square braces [ ] preceding the variable name tell us that we are dealing with arrays.	4.The keyword struct tell us that we are dealing with structures.
5. Initialization of elements can be done during array declaration.	5. Initialization of members can be done only during the structure definition.
6.The elements of an array are store in sequence of	6.The members of a structure are not in sequence of memory

memory location.	location.
7.The array elements are accessed by using index or subscript.	7.The members of a structure are accessed by using dot operator.
8.Its general format is Data type array name[size];  9.Example: int sum[10];	8.Its general format is : Struct <structure name> { Data type member1; Data type member2; . . Data type memberN; }structurevariable(s);  9.Example: Struct student { Int rollno; Char name[10]; }s1;

**Example 1: Write a C program to display int , float and char values using switch case in structure.**

```
void main()
{
union
{
int a;
float b;
char c;
}s;

int ch;
printf("\n 1.int 2.float 3. char");
printf("\n enter the choice");
scanf("%d",&ch);
switch(ch)
{
case 1: printf("\nenter the integer value");
scanf("%d", &s.a);
printf("\n value=%d", s.a);
break;
case 2: printf("\nenter the float value");
```

```

scanf("%f", &s.a);
printf("\n value=%f", s.b);
break;
case 3: printf("\n enter the char");
scanf("%c", &s.a);
printf("\n char=%c", s.c);
break;
}
getch();
}

```

**Output:** 1. Int 2. Float 3. Char

Enter the choice 1

Enter the value 2

Value=2

**Example 2: Write a C program using a structure to create a library catalogue with the following fields.**

**1. Accessno 2.authors' name 3.title 4. Year of publication 5.publisher name 6.price.**

```
void main()
```

```
{
```



```
struct library
{
    int accessno;
    char authername[10];
    char title[10];
    int year;
    char pubname[10];
    float price;
};
```

```
struct library b1={101, "kamthane", "pc&ds" , 2005,
"pearson" , 195.00};
clrscr();
printf("\n access no=%d", b1.accessno);
printf("\n author name=%s",b1.authername);
printf("\n title=%s", b1.title);
printf("\n year=%d", b1.year);
printf("\n pubname=%s", b1.pubname);
printf("\n price=%f", b1.price);
getch();
}
```

**Output:**

```
access no=101
author name=kamthane
title=pc&ds
year=2005
pubname=pearson
price=195.000000_
```

**Example 3: Write a c program to find the topper of the student by using array of structures.**

```
#include<stdio.h>
#include<conio.h>
void main()
{
    struct student
    {
        int rollno, m1,m2,m3,sum;
        char name[10];
        float avg;
    }s[10];

    int i, n, t=0, k=0;
    clrscr();
    printf("\n enter the number of students u want");
    scanf("%d", &n);
    for(i=0;i<n;i++)
```

```

{
printf("\n Enter the name rollno m1 m2 m3");
scanf("\n%s%d%d%d", s[i].name, &s[i].rollno,
&s[i].m1,&s[i].m2,&s[i].m3);
s[i].sum=s[i].m1+s[i].m2+s[i].m3;
s[i].avg=s[i].sum/3.0;
printf("\n sum=%d\n avg=%f\n", s[i].sum, s[i].avg);
}

```

```

t=s[0].avg;
for(i=1;i<n;i++)
{
if(t<s[i].avg)
{
t=s[i].avg;
k=i;
}
}
printf("\n Topper of the class is");
printf("\n-----");

```

```
printf("\n Name=%s\n Rollno=%d\n m1=%d\n m2=%d\n
m3=%d\n ", s[k].name, s[k].rollno, s[k].m1, s[k].m2,
s[k].m3);
printf("\n sum=%d\n avg=%f", s[k].sum, s[k].avg);
getch();
}
```

### Output:

```
enter the number of students u want2
Enter the name rollno m1 m2 m3sai 1 33 33 44
sum=110
avg=36.666668
Enter the name rollno m1 m2 m3kumar 2 33 66 77
sum=176
avg=58.666668
Topper of the class is
-----
Name=kumar
Rollno=2
m1=33
m2=66
m3=77
sum=176
avg=58.666668
```

## UNIT-V

### **Functions & File Handling**

Introduction to Functions, Function Declaration and Definition, Function call Return Types and Arguments, modifying parameters inside functions using pointers, arrays as parameters. Scope and Lifetime of Variables, Basics of File Handling

**6.1 Introduction:** The C language supports two types of functions

1. Library functions or predefined functions
2. User defined functions

**1. Library functions or predefined functions:** The library functions are pre-defined set of functions. Their task is limited. A user cannot understand the internal working of these functions. The user can only use the functions but cannot change or modify them.

Example: `sqrt(81)` gives result 9. Here the user need not worry about its source code, but the result should be provided by the function.

**2. User defined functions:** The user defined functions are totally different. The functions defined by the user according to his/her requirements are called user defined

functions. The user can modify the function according to the requirement. The user certainly understands the internal working of the function.

Definition: A function is a self-contained block is called a function.

[Or]

A subprogram of one or more statements that performs a special task when called.

Use of functions: The use of functions offers flexibility in the design development and implementation of the program to solve complex problems.

**Advantages of functions:**

1. Modular programming.
2. Reduction in the amount of work and development time.
3. Program and function debugging is easier.
4. Division of work is simplified due to the use of divide and conquers principle.
5. Reduction in the size of the program due to code reusability.
6. Logical clarity of the programming will be clear.
7. A library with user defined functions can be created.

**Example: Write a c program on local variables**

```
#include<stdio.h>
#include<conio.h>
void main()
{
    int b=10, c=5;
    clrscr();
    printf("\n In main function b=%d c=%d", b,c);
    fun();
    getch();
}
fun()
{
    int b=20, c=10;
    printf("\n In function b=%d c=%d", b,c);
}
```

**Output:** In main function b=10 c=5

In function b=20 c=10

Example: Write a c program on global variables

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

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

```
int b=10, c=5;
```

```
void main()
```

```
{
```

```
clrscr();
```

```
printf("\n In main function b=%d c=%d", b,c);
```

```
fun();
```

```
b++;
```

```
c--;
```

```
Printf("\n again in main() b=%d c=%d", b,c);
```

```
getch();
```

```
}
```

```
fun()
```

```
{
```

```
b++;
```

```
c--;
```

```
printf("\n In function b=%d c=%d", b,c);
```

```
}
```



**Output:** In main() b=10 c=5

In fun( ) b=11 c=4

Again in main( ) b=12 c=3

**6.2 Function components:** Every function has the following elements associated with it

1. Function declaration and function prototype
2. Function parameters
3. Function definition
4. Return statements
5. Function call

**1. Function declaration and function prototype:** Function is declared as per format given bellow

function-name(arguments/parameter list)

{

local variable declaration;

statement1;

statement2;

return(value);

}

void main()

```

{
.....
abc(x,y,z);           Function call or calling or function
declaration
.....
.....   Actual arguments
}

```

```

abc(l,k,j)           Function definition or called function
{
.....   Formal arguments
.....
return();           return value
}

```

**2. Function parameters:** The parameters specified in the function call (calling) are known as actual parameters and those specified in the function definition (called) are known as formal parameters.

```

void main()
{

```

```

int x=1, y=2 ,z;
z=add(x,y);      Function call(calling)
printf("z=%d", z);
getch();
}
add(a,b)      Function definition (called)
{
    return(a+b);
}

```

**3. Function definition:** The function itself is referred to as function definition. The first line of the function definition is known as function declaratory and is followed by the function body.

```

add(a,b)      Function definition (called)
{
    return(a+b);
}

```

**4. Return statements:** Functions can be grouped into two categories

1. Functions that do not have a return value(void)
- 2.Functions that have a return value.

void main()                    The main() function should not return anything.

```
{  
.....  
.....  
getch();  
}
```

int main()                    The main() function should return a integer type of value.

```
{  
.....  
.....  
return(0);  
getch();  
}
```

**5. Function call:** A function call is specified by the function name followed by the arguments enclosed in parenthesis and terminated by a semicolon. The return type is not mentioned in the function call.

Example: z=add(x, y);            Function call(calling)

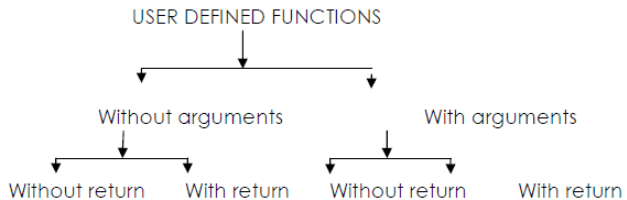
### 6.3 Variables: There are two kinds of variables.

1. Local variables
2. Global variables

<b>Local variables</b>	<b>Global variables</b>
<p>1. It is declaration inside the function.</p> <p>2.It is declared by <code>auto int a=10</code> or <code>int a = 10</code> (auto is default scope)</p> <p>3. If it is not initialized garbage value is stored. EX: <code>int a;</code> a is garbage value.</p> <p>4. It is created when the function starts execution and lost when the function terminates</p> <p>5. It is visible in only one function.</p> <p>6. It can be accessed in only one function that is the function where it is declared.</p> <p>7. Data sharing is not possible that is data of local variable</p>	<p>1.It is declaration outside the function</p> <p>2. It is declared by <code>int a=10</code></p> <p>3. If it is not initialized zero is stored EX: <code>int a; a = 0.</code></p> <p>4. It is created before program execution starts and lost when program terminates</p> <p>5. It is visible throughout the program.</p> <p>6. It can be accessed in more than one function.</p> <p>7. Data sharing is possible that is multiple functions can access the same global variable.</p>

<p>can be accessed by only one function</p> <p>8. Parameters passing is required for local variable that is local variable of one variable</p> <p>9. If value of local variable is modified in one function changes are not visible in another functions</p>	<p>8. Parameters passing is not required for global variable since global variable is visible throughout the program.</p> <p>9. If value of global variable is modified in one function changes are visible in rest of the program.</p>
--	---

**6.4 Types of functions:** Depending upon the argument present, return value sends the result back to the calling function based on this the functions are divided into four types.



**Example:**

- `void sum( )`            function without arguments and without return value
- `int sum( )`            function without arguments and with return value
- `void sum(int ,int)`    function with arguments and without return value
- `int sum(int,int)`     function with arguments and with return value

**Example1: Write a C program on without arguments and return values**

```
#include<stdio.h>
#include<conio.h>
void main()
{
void message();
}
void message()
{
printf("Have a nice day");
}
```

**Output:** Have a nice day

**Explanation:** - This program contains a user defined function named message (). It requires no arguments and returns nothing. It displays only a message when called.

**Example2: Write a C program on with arguments without return values**

```
#include<stdio.h>
#include<conio.h>
void main()
{
int j=0;
void sqr();
for(j=1;j<=5;j++)
sqr(j);
}
void sqr(int k)
{
printf("\n %d",k*k);
}
```

**Output:** 1 4 9 16 25

Explanation: Here the main () function passes one argument per call to the function sqr(). The function sqr() collects this arguments and prints its square. The function sqr() is void.



### **Example 3: Write a C program on with arguments and with return values**

```
#include<stdio.h>
#include<conio.h>
void main()
{
int date(int, int,int);
int d,m,y,t;
clrscr();
printf("enter date dd/mm/yy");
scanf("%d%d%d",&d,&m,&y);
t=date(d,m,y);
printf("\n Tomorrow=%d/%d/%d",d,m,y);
return 0;
}
date(int x, int y, int z)
{
printf("\n Today=%d/%d/%d",x,y,z);
return(++x);
}
```

**output:** enter date dd/mm/yy 12 12 12

Today = 12/12/12

Tomorrow= 13/12/12

**Explanation:** In the above program three values date, month, and year are passed to functions date (). The function displays the date. The function date () returns the next date. The next date is printed in function main (). Here, function date () receives arguments and returns the values.

**Example 4: Write a c program on without arguments and without returns values**

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

main()
{
int sum(),a,s;
clrscr();
s=sum();
printf("sum=%d", s);
return 0;
}

sum()
{
int x,y,z;
```

```
printf("\n Enter the three values");  
scanf("%d %d %d", &x,&y,&z);  
return(x+y+z);  
}
```

**Output:** enter the three values 3 5 4

Sum=12

**6.5 Parameter passing mechanism:** There are two parameter passing mechanisms

1. Call by value. Or passing by value
2. Call by reference or passing by address or call by address.

**1. Call by value:**

Passing arguments by value means, the contents of the arguments in the calling function are not changed, even if they are changed in the called function. This is because the content of the variable is copied to the formal parameter of the function definition, thus preventing the contents of the argument in the calling function.

**2. Call by reference:**

Call by reference means sending the address of variables as arguments to the Function. When addresses are sent , the

changes occurred in the called function can also effect in the calling function.

**Example 1: Write a C program to perform parameter passing mechanism by using call by value.**

```
#include<stdio.h>
#include<conio.h>
void add(int);
void main()
{
    int x=5;
    printf("Before x=%d",x);
    add(x);
    printf("After x=%d",x);
    getch();
}
void add(int y)
{
    y=y+5;
    printf("In called function x=%d",y);
}
```

**Output:**

Before x=5.

In called function x=10.

After x=5.

**Advantages:** 1. Expression can be passed as arguments.  
2. Un wanted changes to the variables in the calling function can be avoided.

**Disadvantages:** Information cannot be passed back from the calling function to the called function through arguments.

**Example 2: Write a C program to perform parameter passing mechanism by using call by reference.**

```
#include<stdio.h>
#include<conio.h>
void add(int *);
void main()
{
    int x=5;
    printf("Before x=%d",x);
    add(&x);
    printf("After x=%d",x);
    getch();
}
void add(int *y)
```

```
{  
    *y=*y+5;  
    printf("In called function x=%d", *y);  
}
```

### **Output:**

Before x=5.

In called function x=10.

After x=10.

**6.6 Recursion:** A function which calls itself until a certain condition is reached is called recursive function. The recursion can be

1. Direct recursion    2. Indirect recursion.

**1. Directive recursion:** The direct recursion function calls itself till the condition is true.

**2. Indirect recursion:** In indirect recursion a function calls another function then the called function calls the calling function.

**Example1: Write a C program to find the factorial of a given number using recursion**

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

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

```
long int fact(int);
void main()
{
    long int f;
    int n;
    clrscr();
    printf("Enter the number");
    scanf("%d",&n);
    f=fact(n);
    printf("The factorial of a given number is %ld", f);
    getch();
}
```

```
long int fact(int x)
{
    long int m;
    if(x==1|| x==0)
        return(1);
    else
    {
        m=x*fact(x-1);
        return(m);
    }
}
```

```
}  
}
```

**Output:** Enter the number 5

The factorial of a given number is 120

**Example 2: Write a C program to find the Fibonacci series using recursion**

```
#include<stdio.h>  
#include<conio.h>  
int fib(int);  
void main()  
{  
    int i, n;  
    clrscr();  
    printf("Enter the number");  
    scanf("%d",&n);  
    printf("The fibnocci series is");  
    for(i=1;i<=n;i++)  
        printf("%d", fib(i));  
    getch();  
}  
int fib(int x)
```



```
{  
    if(x==1|| x==2)  
        return(x-1);  
    else  
        return(fib(x-1)+fib(x-2));  
}
```

**Output:** Enter the number 5

The Fibonacci series is 0 1 1 2 3

**6.7 STORAGE CLASSES:** Automatic, External, Static and Register Variables.

Normally the life of a variable is limited to a function as long as the function is alive. How to make it alive in a file or throughout the program or limiting only to a block inside a function or to make common to a desired couple of functions etc., The answer lies in “STORAGE CLASSES” or “VARIABLE TYPES”.

There are four types of storage classes.

1. Automatic variables.
2. External variables.
3. Static variables.
4. Register variables.

## 1. Automatic variables:

Automatic variables are declared inside a function, in which they are to be utilized. They are created when the function is called and destroyed automatically when the function is exited. By default all variables are automatic variables.

These are also called local or internal variables.

Ex. Write a C Program on Automatic Variables.

```
#include<stdio.h>
#include<conio.h>
Void main()
{
int a=9;
clrscr();
fun1();
printf("\n in main() a=%d",a);
getch();
}
fun1()
{
Auto int a=8;
fun2()
{
```

```
int a =7;
printf("\n in fun2() a=%d",a);
}
```

### **Output:**

In fun2 () a=7

In fun1 () a=8

In main () a=9

### **2. Register variables:**

1. This is local to a function or a block.
2. If a compiler finds a physical register in the CPU free for the time being, and also big enough to hold the value, then it may stick that variable in that register. Otherwise the compiler treats that variable as ordinary.
3. It is machine dependent. But compiler will not give error messages even if no register available in reserve.

### **Ex: Write a C program on Register variables.**

```
#include<stdio.h>
#include<conio.h>
Void main()
{
register int a=1;
```

```
clrscr();  
for( ;a<=3;a++)  
printf(“\na=%d”,a);  
getch();  
}
```

Output:

```
a=1          a=2          a=3
```

### 3. Static variables:

When a variable is declared as a static variable it is assigned the value zero. Static variables are initialized only once.

They will not be initialized for second time during the program. When static is applied to a global variable, the global variable becomes inaccessible outside the file.

**Ex: Write a C program on Static Variables.**

```
#include<stdio.h>  
#include<conio.h>  
void main()  
{  
int a;  
static int b;  
clrscr();  
printf(“\na=%d\nb=%d”,a,b);
```

```
getch();  
}
```

Output:

```
a=25338,      b=0;
```

#### **4. External variables:**

Variables that are both alive and active throughout the entire program are called external variables. These variables are available to all functions in that program. Whatever changes that occur in a function, will affect the value of that variable.

Ex: Write a C program on External Variables.

```
#include<stdio.h>  
#include<conio.h>  
int a=5;  
void main()  
{  
clrscr();  
fun1();  
fun2();  
printf("\n In main() a=%d",a);  
getch();  
}  
fun1()
```

```
{
printf("\n in fun1() a=%d",a);
}
fun2()
{
Printf("\n in fun2() a=%d",a);
}
```

**Output:**

In fun1 () a=5

In fun2 () a=5

In main () a=5

## Difference between Iterative statements and Recursion

<b>Iterative statements</b>	<b>Recursion</b>
<p>1. Function calls some other functions.</p> <p>2. while loop, do while loop or for loop is necessary in iterative function.</p> <p>3. Fast in execution since one function call is enough to get the result.</p> <p>4. Function call leads to value Eg: fact(4) = 24.</p> <p>5. We require mathematical steps or procedure to write an iterative function.</p> <p>6. Stacks are not used during execution</p>	<p>1. Function calls some itself.</p> <p>2. if statement is necessary in recursive Function</p> <p>3. Slow in execution since several function calls are involved to get the result, as number of function increases execution will slow down.</p> <p>4. Function call leads to another function call Eg: fact(4) = 4 xfact(3)</p> <p>5. We require a formula to write a recursive function.</p> <p>6. Stacks are used during execution of recursive functions.</p>

**FILES:** Introduction, File Types, Basic operations on Files, File I/O, Command Line Arguments

**6.8 Introduction:** A file contains data/information which is stored permanently in a storage device. Generally used storage devices are CDs, DVDs & hard disks. When large quantity of data is required to be stored and processed, the concept of file is used.

A file stored in a storage device is always identified using a name(e.g. student. Dat, info.txt). Normally a filename has a primary name and a secondary name, which are separated by a dot(.).

STUDENT. DAT

Primary name separator secondary name

**Definition:** File is a set of records that can be accessed through the set of library functions.

**Streams:** Stream means reading and writing of data. The streams are designed to allow the user to access the files efficiently. A stream is a file or physical device like keyboard, printer and monitor.



## 6.9 Types of files: Based on the type of data

1. Data file
2. Text file

Based on the accessing the data

1. Sequential file
2. Random file

**1. Data file:** A data file contains data stored in the form of records. A record is a collection of data related to a person or item. For example, a student record may contain data like roll number, student name and marks obtained by him. A file contain may such records. Imagine the records are arranged one by one as shown in figure.

A data file with records

2201	arjun	78
2202	arya	80
.		
.		
.		
2249	shekar	90
2250	swamy	99

**2. Text file:** A text file contains information stored in the form of string characters. The characters entered through the keyboard are stored continuously as illustrated bellow.

**1. Sequence file:** In sequential file data/information is stored sequentially one by another. The data is read in the same order in which they are stored.

**2. Random access file:** In random access file the data/information can be read randomly. Normally a key is used to identify the required record in random file accessing.

### **6.10 Basic operations on Files**

**File declaration:** A file is declared and the data is accessed using a file pointer. It has the following general form.

Various functions used in the file operations.

<b>Function</b>	<b>Operation</b>
fopen( )	Creates a new file for read/write operation.
fclose()	Closes a file associated with the pointer.
closeall( )	Closes all opened files with fopen ( ).
fgetc( )	Reads the character from current pointer position and advances the pointer to next character.
getc( )	Same as fgetc ( ).
fprintf( )	Writes all types of data values to the file.
fscanf( )	Reads all types of data values from a file.

putc( )	Writes character one by one to a file.
fputc( )	Same as putc( ).
gets( )	Reads string from the file.
puts( )	Writes string to the file.
putw( )	Writes an integer to the file.
getw( )	Reads an integer from the file.
fread( )	Reads structured data written by fwrite ( ).
fwrite( )	Writes block of structured data to the file.
fseek( )	Sets the pointer position anywhere in the file.
feof( )	Detects the end of file.
ferror( )	Reports error occurred while read/write operations.
perror( )	Prints compilers error messages along with user defined messages.
ftell( )	Returns the current pointer position.

rewind( )	Sets the record pointer at the beginning of the file.
unlink ( )	Removes the specified file from the disk.
remove( )	Removes the specified file from the disk changes the name of the file.

**6.11 The file pointer (fp):** A file pointer is a pointer to a structure of type FILE. It points to information that defines various things about the file, including its name, status and the current position of the file.

Example: FILE \*fp;

**6.12 fopen( ) function:** The fopen ( ) function is used to open a file and set the file pointer to the beginning/end of a file. It has the following form.

**Syntax:**

fp= fopen(“filename”, “mode”);

Where fp refers to the name of the file to be opened. Mode refers to the operation mode to access data. The following “mode” are used in data processing.

<b>Mode</b>	<b>Meaning</b>
r	Open a text file for reading.
w	Create a text file for writing.
a	Append to a text file.
rb	Open a binary file for reading.
wb	Creates a binary file for writing.
ab	Append to a binary file.
r+	Open a text file for read/write.
w+	Create a text file for read/write.
a+	Append for create a text file for read/write.
r+b	Open a binary file for read/write.
w+b	Create a binary file for read/write.
a+b	Append or create a binary file for read/write.

**6.1 3 fclose( ) function:** All files that are opened should be closed after all input and output operations with the file. This is to prevent data from getting corrupted. fclose() function is used to close an active file. It has the following form.

**Syntax:**

```
int fclose (FILE *fp);
```

**Example:** fclose(fp);

Where fp refers to file pointer. The function return EOF if an error occurs then uses the standard function ferror ( ).

**Example1: Write a C program to read the data from the keyboard, write it to a file called input, again read the same data from the input file, and displaying it on the screen.**

```
#include<stdio.h>

void main()
{
FILE *fp;
int ch;

/* to store data in a file */
printf("\nEnter the data");
fp=fopen("input.dat", "w");
while((ch=getchar())!=EOF)
putc(ch, fp);
fclose(fp);

/* to display data on the screen */
printf("\nThe data stored in input.dat file");
fp=fopen("input.dat", "r");
```

```
while((ch=getc(fp))!=EOF)
printf("%d",ch);
fclose(fp);
getch();
}
```

Output: welcome to Nbkrist

**Example2: To write a C program to read in a line of lower case text from a file and display its upper case equivalent on the screen.**

```
#include<stdio.h>
void main()
{
FILE *fp;
int ch;
clrscr();
fp=fopen("sample.txt", "r");
while((ch=getc(fp))!=EOF)
putchar(toupper(ch));
fclose(fp);
getch();
}
```

Output: first to create the file sample.txt and then write the text "hai"

The output is going to show HAI

**Example: Write a C program to count chars, spaces, tabs and new lines in a file.**

```
#include<stdio.h>
#include<conio.h>
void main()
{
FILE *fp;
int ch;
int nol=1,not=0,noc=0,nob=0;
fp=fopen("srist.txt", "r");
while(1)
{
ch=getc(fp);
if(ch==EOF)
break;
noc++;
if(ch==' ')
nob++;
if(ch=='\n')
```



```

    nol++;
    if(ch=='\t')
        not++;
}
fclose(fp);
printf("\n number of characters=%d",noc);
printf("\n number of blanks=%d",nob);
printf("\n number of tabs=%d",not);
printf("\n number of lines=%d",nol);
getch();
}

```

**Output:** First to create the file srist.txt and then write  
welcome to Nbkrist Vidyanagar

Number of characters=26

Number of blanks=3

Number of tabs=1

Number of lines=1

## File I/O

**6.14 fscanf( ):** The fscanf( ) function is used to read data from a file. It is similar to the scanf( ) function except that fscanf( ) is used to read data from disk. It has the following form.

Syntax:                fscanf(fp,                “format        string”,  
&v1,&v2,.....&vn);

**6.55 fprintf( ):** The fprintf( ) function is used to write data to a file. It is similar to printf( ) function except that fprintf( ) is used to write data to the disk. It has the following form.

Syntax: fprintf(fp, “format string” , v1,v2, .....vn);

**Example: Write a C program to create a file “student.txt” , contains information such as student roll number , name, total marks.**

```
#include<stdio.h>
#include<conio.h>
void main()
{
    FILE *fp;
    int i, n, rno, total;
    char sname[20];
    fp=fopen("student.txt", "w");
    printf("\n how many students");
    scanf("%d",&n);
    for(i=0;i<n;i++)
    {
```

```

printf("\n Enter rno, sname, total");
scanf("%d %s %d", &rno,sname,&total);
fprintf(fp, "%d %s %d",rno, sname,total);
}
fclose(fp);
getch();
}

```

**6.16 Random access files:** Random access files can be referred randomly. For this separate functions are available. The functions that are used in random access files are

1. fseek( ),
2. ftell( ),
3. rewind( ).

fseek( ) function: The fseek( ) function is used to move the file pointer to any position in a file from a given reference position. It has the following form.

Syntax: fseek(fp, n, position);

Where “fp” is a file pointer “n”is a + or – long integer number that represent the number of bytes to be skipped and “position” is the position from which n bytes to be skipped.

0→ beginning (SEEK\_SET)

1→current (SEEK\_CUR)

2→end (SEEK\_END)

Example: `fseek(fp, 10, 0);` or `fseek(fp, 10, SEEK_SET);`

The file pointer is repositioned in the forward direction by 10 bytes.

**Example: Write a C program to illustrate `fseek()`**

```
#include<stdio.h>  
void main()  
{  
    FILE *fp;  
    int n;  
    clrscr();  
    fp=fopen("data.txt", "r");  
    fseek(fp,4L, 0);  
    n=getc(fp);  
    while(n!=EOF)  
    {  
        putchar(n);  
        n=getc(fp);  
    }  
    fclose(fp);  
    getch();  
}
```

**6.17 ferror( ):** The ferror( ) function determines whether a file operation has produced an error.

**Syntax:** int ferror(FILE \*fp);

Where fp is a file pointer. It returns true if an error has occurred during the last file operation. Otherwise it returns false.

**6.18 Writing a character ( ):** The putc( ) function writes characters to a file that was previously opened for writing using the fopen( ).

**Syntax:** int putc(int ch, FILE \*fp);

If a putc( ) function operation is successful, it returns the character written. Otherwise, it returns EOF.

**6.19 Reading a character:** The function getc( ) reads characters from a file opened in read mode by fopen( ).

**Syntax:** int getc(FILE \*fp);

Where fp is file pointer of type FILE returned by fopen( ).

**6.20 feof( ):** The C file system includes the function feof( ), which determines when the end of the file has been encountered.

**Syntax:** int feof ( FILE \*fp);

feof( ) returns true if the end of the file has been reached. Otherwise it returns 0(zero).

**6.21 Command line arguments:** The function main( ) in C can also pass arguments or parameters like the other functions. It has two arguments argc (for argument count) and argv (for argument vector). It may have one of the following two forms.

```
void main(int argc, char *argv[ ])
```

or

```
void main(int argc, char **argv[ ])
```

Where argc represents the number of arguments, argv is a pointer to an array of strings or pointer to pointer to character.

The argument argv is used to pass strings to the programs. Hence the arguments argc and argv are called as program parameter.

The program name is then interpreted as an operation system command. Hence the line in which it appears is generally referred to as a command line.

**Example:** Program-name parameter1, parameter2  
.....parameter n

The individual items must be separated from one another either by blank space or by tabs. Some operating systems permit blank space to be included within a parameter provided the entire parameter is enclosed in quotation mark.

**Example 1:** sample red white blue

Argc= 4

Argv[0]= sample.exe

Argv[1]=red

Argv[2]= white

Argv[3]= blue

Example 2: sample red "white blue"

Argc= 3

Argv[0]= sample.exe

Argv[1]=red

Argv[2]= white blue

**Example: Write a C program on command line argument**

```
#include<stdio.h>
#include<conio.h>
void main(int argc, char *argv[ ])
{
int count;
printf("argc=%d\n", argc);
for(count=0; count<argc; ++count)
printf("argv[%d]=%s\n", count, argv[count]);
getch();
}
```

**Output:** go to the command prompt

