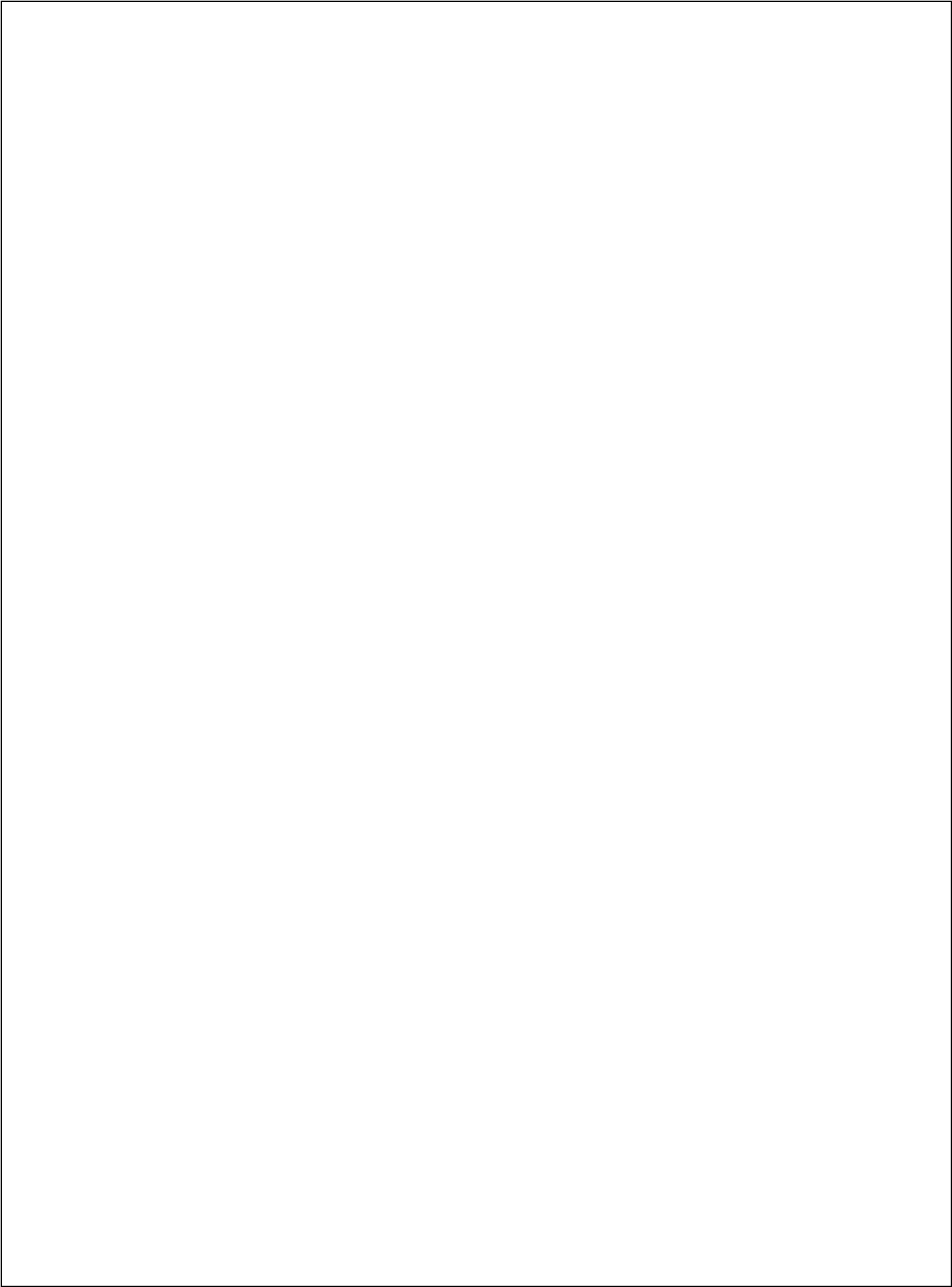


PROGRAMMING FOR PROBLEM SOLVING

—Riyaz Mohammad—



CS103ES/CS203ES: PROGRAMMING FOR PROBLEM SOLVING

B.Tech. I Year I Sem.

L	T	P	C
3	1	0	4

Unit - 1: Introduction to Programming

Introduction to components of a computer system: disks, primary and secondary memory, processor, operating system, compilers, creating, compiling and executing a program etc., Number systems

Introduction to Algorithms: steps to solve logical and numerical problems. Representation of Algorithm, Flowchart/Pseudocode with examples, Program design and structured programming.

Introduction to C Programming Language: variables (with data types and space requirements), Syntax and Logical Errors in compilation, object and executable code, Operators, expressions and precedence, Expression evaluation, Storage classes (auto, extern, static and register), type conversion, The main method and command line arguments

Bitwise operations: Bitwise AND, OR, XOR and NOT operators

Conditional Branching and Loops: Writing and evaluation of conditionals and consequent branching with if, if-else, switch-case, ternary operator, goto, Iteration with for, while, do- while loops

I/O: Simple input and output with scanf and printf, formatted I/O, Introduction to stdin, stdout and stderr.

Command line arguments

Unit - II: Arrays, Strings, Structures and Pointers:

Arrays: one and two dimensional arrays, creating, accessing and manipulating elements of arrays

Strings: Introduction to strings, handling strings as array of characters, basic string functions available in C (strlen, strcat, strcpy, strstr etc.), arrays of strings

Structures: Defining structures, initializing structures, unions, Array of structures

Pointers: Idea of pointers, Defining pointers, Pointers to Arrays and Structures, Use of Pointers in self-referential structures, usage of self referential structures in linked list (no implementation)

Enumeration data type

Unit - III: Preprocessor and File handling in C:

Preprocessor: Commonly used Preprocessor commands like include, define, undef, if, ifdef, ifndef

Files: Text and Binary files, Creating and Reading and writing text and binary files, Appending data to existing files, Writing and reading structures using binary files, Random access using fseek, ftell and rewind functions.

Unit - IV: Function and Dynamic Memory Allocation:

Functions: Designing structured programs, Declaring a function, Signature of a function, Parameters and return type of a function, passing parameters to functions, call by value, Passing arrays to functions, passing pointers to functions, idea of call by reference, Some C standard functions and libraries

Recursion: Simple programs, such as Finding Factorial, Fibonacci series etc., Limitations of Recursive functions

Dynamic memory allocation: Allocating and freeing memory, Allocating memory for arrays of different data types

Unit - V: Introduction to Algorithms:

Algorithms for finding roots of quadratic equations, finding minimum and maximum numbers of a given set, finding if a number is prime number, etc.

Basic searching in an array of elements (linear and binary search techniques),

Basic algorithms to sort array of elements (Bubble, Insertion and Selection sort algorithms), Basic concept of order of complexity through the example programs

TEXT BOOKS:

1. Byron Gottfried, Schaum's Outline of Programming with C, McGraw-Hill
2. B.A. Forouzan and R.F. Gilberg C Programming and Data Structures, Cengage Learning, (3rd Edition)

REFERENCE BOOKS:

1. Brian W. Kernighan and Dennis M. Ritchie, The C Programming Language, Prentice
2. Hall of India
3. R.G. Dromey, How to solve it by Computer, Pearson (16th Impression)
4. Programming in C, Stephen G. Kochan, Fourth Edition, Pearson Education.
5. Herbert Schildt, C: The Complete Reference, Mc Graw Hill, 4th Edition

UNIT-I INTRODUCTION TO COMPUTERS

COMPUTER SYSTEMS

“A Computer is an electronic device that stores, manipulates and retrieves the data.”

We can also refer computer computes the information supplied to it and generates data.

A System is a group of several objects with a process. For Example: Educational System involves teacher, students (objects). Teacher teaches subject to students i.e., teaching (process). Similarly a computer system can have objects and process.

The following are the objects of computer System

- a) User (A person who uses the computer)
- b) Hardware
- c) Software

Hardware: Hardware of a computer system can be referred as anything which we can touch and feel.

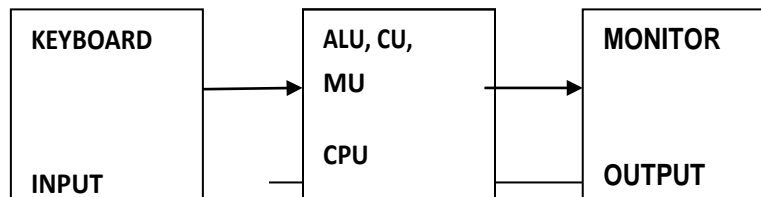
Example : Keyboard and Mouse etc.

The hardware of a computer system can be classified as

Input Devices(I/P)

Processing Devices (CPU)

Output Devices(O/P)



CPU: Alternately referred to as a processor, central processor, or microprocessor,

the CPU is the central processing unit of the computer.

A computer's CPU handles all instructions it receives from hardware and software running on the computer.

ALU: It performs the Arithmetic and Logical Operations such as +,-,*,/

(Arithmetic Operators) &&, || (Logical Operators)

CU: Every Operation such as storing, computing and retrieving the data should be governed by the control unit.

MU: The Memory unit is used for storing the data.

The Memory unit is classified into two types.

They are 1) Primary Memory

2) Secondary Memory

1) Primary memory: The following are the types of memories which are treated as primary

ROM: It represents Read Only Memory that stores data and instructions even when the computer is turned off (non volatile). The Contents in the ROM can't be modified once if they are written. It is used to store the BIOS information.

RAM: It represents Random Access Memory that stores data and instructions when the computer is turned on and data is erased when the computer is turned off (volatile). The contents in the RAM can be modified any no. of times by instructions. It is used to store the programs under execution.

Cache memory: It is a special type of internal memory used by many central processing units to increase their performance or "throughput". Some of the information in the main memory is duplicated in the cache memory, which is slightly slower but of much greater capacity than the processor registers, and faster but much smaller than main memory.

2) Secondary Memory: The following are the different kinds of memories

Magnetic Storage: The Magnetic Storage devices store information that can be read, erased and rewritten a number of times.

Example: Floppy Disks, Hard Disks, Magnetic Tapes

Optical Storage: The optical storage devices that use laser beams to read and write stored data.

Example: CD (Compact Disk), DVD (Digital Versatile Disk)

COMPUTER SOFTWARE

Software of a computer system can be referred as anything which we can feel and see.

Example: Windows, icons

Computer software is divided in to two broad categories: system software and application software .System software manages the computer resources .It provides the interface between the hardware and the users. Application software, on the other hand is directly responsible for helping users solve their problems.

System Software

System software consists of programs that manage the hardware resources of a computer and perform required information processing tasks. These programs are divided into three classes: the operating system, system support, and system development.

The **operating system** provides services such as a user interface, file and database access, and interfaces to communication systems such as Internet protocols. The primary purpose of this software is to keep the system operating in an efficient manner while allowing the users access to the system.

System support software provides system utilities and other operating services. Examples of system utilities are sort programs and disk format programs. Operating services consists of programs that provide performance statistics for the operational staff and security monitors to protect the system and data.

The last system software category, **system development software**, includes the language translators that convert programs into machine language for execution ,debugging tools to ensure that the programs are error free and computer –assisted software engineering(CASE) systems.

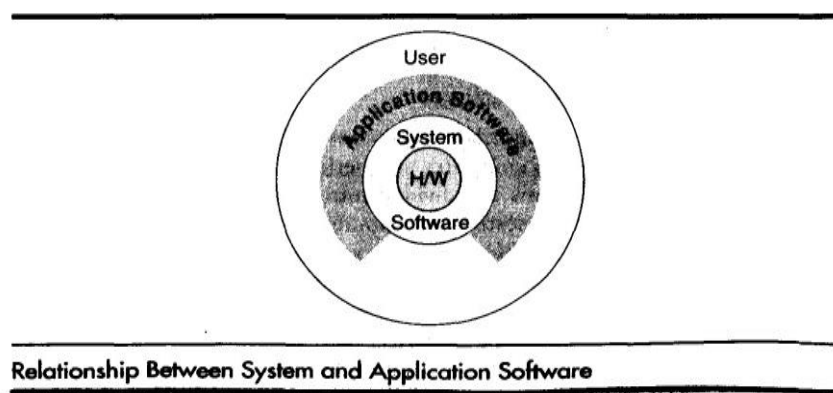
Application software

Application software is broken in to two classes: general-purpose software and application – specific software. **General purpose software** is purchased from a software developer and can be used for more than one application. Examples of general purpose software include word processors, database management systems, and computer aided design systems. They are labeled general purpose because they can solve a variety of user computing problems.

Application –specific software can be used only for its intended purpose.

A general ledger system used by accountants and a material requirements planning system used by a manufacturing organization are examples of application-specific software. They can be used only for the task for which they were designed they cannot be used for other generalized tasks.

The relationship between system and application software is shown below. In this figure, each circle represents an interface point .The inner core is hard ware. The user is represented by the out layer. To work with the system, the typical user uses some form of application software. The application software in turn interacts with the operating system, which is a part of the system software layer. The system software provides the direct interaction with the hard ware. The opening at the bottom of the figure is the path followed by the user who interacts directly with the operating system when necessary.



INTRODUCTION TO C PROGRAMMING LANGUAGES

To write a program (tells what to do) for a computer, we must use a computer language. Over the years computer languages have evolved from machine languages to natural languages. The following is the summary of computer languages

1940's	--	Machine Languages
1950's	--	Symbolic Languages
1960's	--	High Level Languages

Machine Language

In the earliest days of computers, the only programming languages available were machine languages. Each computer has its own machine language which is made of streams of 0's and 1's. The instructions in machine language must be in streams of 0's and 1's. This is also referred as binary digits. These are so named as the machine can directly understand the programs

Advantages:

- 1) High speed execution
- 2) The computer can understand instructions immediately
- 3) No translation is needed.

Disadvantages:

- 1) Machine dependent
- 2) Programming is very difficult
- 3) Difficult to understand
- 4) Difficult to write bug free programs
- 5) Difficult to isolate an error

Example Addition of two numbers

2	0 0 1 0
+ 3	0 0 1 1
---	-----
5	0 1 0 1
---	-----

Symbolic Languages (or) Assembly Language

In the early 1950's Admiral Grace Hopper, a mathematician and naval officer, developed the concept of a special computer program that would convert programs into machine language.

These early programming languages simply mirrored the machine languages using symbols or mnemonics to represent the various language instructions. These languages were known as symbolic languages. Because a computer does not understand symbolic language it must be translated into the machine language. A special program called an **Assembler** translates symbolic code into the machine language. Hence they are called as Assembly language.

Advantages:

- 1) Easy to understand and use
- 2) Easy to modify and isolate error
- 3) High efficiency
- 4) More control on hardware

Disadvantages:

- 1) Machine Dependent Language
- 2) Requires translator
- 3) Difficult to learn and write programs
- 4) Slow development time
- 5) Less efficient

Example:

2	PUSH 2,A
3	PUSH 3,B
+	ADD A,B
5	PRINT C

High-Level Languages

The symbolic languages greatly improved programming efficiency they still required programmers to concentrate on the hardware that they were using working with symbolic languages was also very tedious because each machine instruction had to be individually coded. The desire to improve programmer efficiency and to change the focus from the computer to the problems being solved led to the development of high-level languages.

High-level languages are portable to many different computers allowing the programmer to concentrate on the application problem at hand rather than the intricacies of the computer.

C	A systems implementation Language
C++	C with object oriented enhancements
JAVA	Object oriented language for internet and general applications using basic C syntax

Advantages:

- 1) Easy to write and understand
- 2) Easy to isolate an error
- 3) Machine independent language
- 4) Easy to maintain
- 5) Better readability
- 6) Low Development cost
- 7) Easier to document
- 8) Portable

Disadvantages:

- 1) Needs translator
- 2) Requires high execution time
- 3) Poor control on hardware
- 4) Less efficient

Example: C language

```
#include<stdio.h>
void main()
{
    int a,b,c;
    scanf("%d%d",&a,&b);
    c=a+b;
    printf("%d",c);
}
```

Difference between Machine, Assembly, High Level Languages

Feature	Machine	Assembly	High Level
Form	0's and 1's	Mnemonic codes	Normal English
Machine Dependent	Dependent	Dependent	Independent
Translator	Not Needed	Needed(Assembler)	Needed(Compiler)
Execution Time	Less	Less	High
Languages	Only one	Different Manufacturers	Different Languages
Nature	Difficult	Difficult	Easy
Memory Space	Less	Less	More

LANGUAGE TRANSLATORS

These are the programs which are used for converting the programs in one language into machine language instructions, so that they can be executed by the computer.

- 1) **Compiler:** It is a program which is used to convert the high level language programs into machine language
- 2) **Assembler:** It is a program which is used to convert the assembly level language programs into machine language
- 3) **Interpreter:** It is a program; it takes one statement of a high level language program, translates it into machine language instruction and then immediately executes the resulting machine language instruction and so on.

Comparison between a Compiler and Interpreter

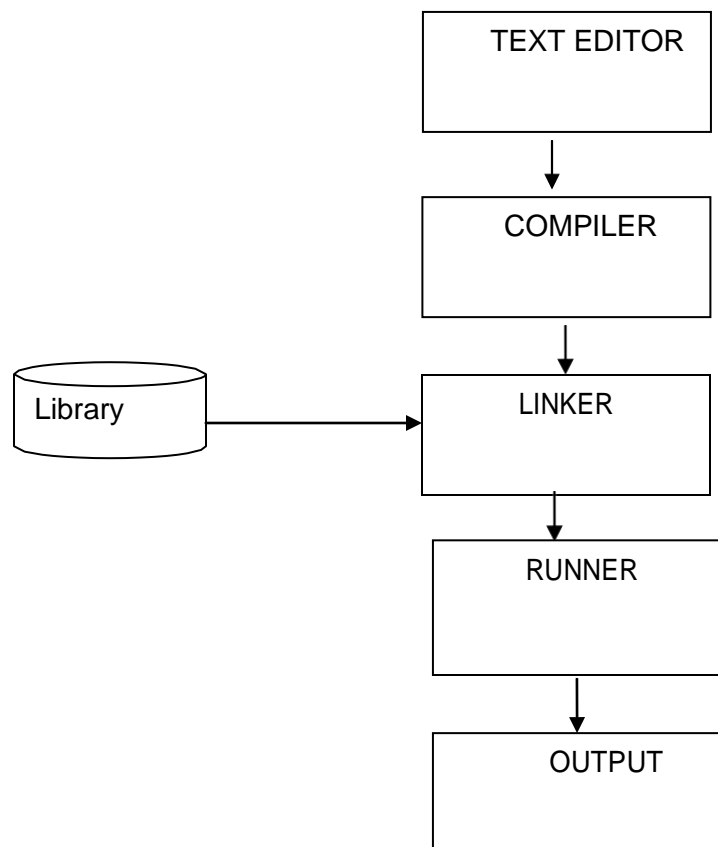
COMPILER	INTERPRETER
A Compiler is used to compile an entire program and an executable program is generated through the object program	An interpreter is used to translate each line of the program code immediately as it is entered
The executable program is stored in a disk for future use or to run it in another computer	The executable program is generated in RAM and the interpreter is required for each run of the program
The compiled programs run faster	The Interpreted programs run slower
Most of the Languages use compiler	A very few languages use interpreters.

CREATING AND RUNNING PROGRAMS

The procedure for turning a program written in C into machine Language. The process is presented in a straightforward, linear fashion but you should recognize that these steps are repeated many times during development to correct errors and make improvements to the code.

The following are the four steps in this process

- 1) Writing and Editing the program
- 2) Compiling the program
- 3) Linking the program with the required modules
- 4) Executing the program



Sl. No.	Phase	Name of Code	Tools	File Extension
1	TextEditor	Source Code	C Compilers Edit, Notepad Etc..,	.C
2	Compiler	Object Code	C Compiler	.OBJ
3	Linker	Executable Code	C Compiler	.EXE
4	Runner	Executable Code	C Compiler	.EXE

WRITING AND EDITING PROGRAMS

The software used to write programs is known as a text editor. A text editor helps us enter, change and store character data. Once we write the program in the text editor we save it using a filename stored with an extension of .C. This file is referred as source code file.

Compiling Programs

The code in a source file stored on the disk must be translated into machine language. This is the job of the compiler. The Compiler is a computer program that translates the source code written in a high-level language into the corresponding object code of the low-level language. This translation process is called *compilation*. The entire high level program is converted into the executable machine code file. The Compiler which executes C programs is called as C Compiler. Example Turbo C, Borland C, GC etc.

The C Compiler is actually two separate programs:

The Preprocessor
the Translator

The Preprocessor reads the source code and prepares it for the translator. While preparing the code, it scans for special instructions known as preprocessor commands. These commands tell the preprocessor to look for special code libraries. The result of preprocessing is called the translation unit.

After the preprocessor has prepared the code for compilation, the translator does the actual work of converting the program into machine language. The translator reads the translation unit and writes the resulting object module to a file that can then be combined with other precompiled units to form the final program. An object module is the code in the machine language.

Linking Programs

The Linker assembles all functions, the program's functions and system's functions into one executable program.

Executing Programs

To execute a program we use an operating system command, such as run, to load the program into primary memory and execute it. Getting the program into memory is the function of an operating system program known as the **loader**. It locates the executable program and reads it into memory. When everything is loaded the program takes control and it begins execution.

NUMBER SYSTEM: It is used to define magnitude of any quantity.

Types	Base or Radix(r)
1. Binary	Radix (r)=2 (0 to r-1) = 0 to 2-1 = 0,1
2. Octal	Radix (r) = 8 (0 to r-1) = 0 to 8-1 = 0,1,2,.....7
3. Decimal	Radix (r)= 10 (0 to r-1) = 0 to 10 -1 = 0,1,2,.....9
4. Hexadecimal	Radix (r) = 16 (0 to r-1) = 0 to 16-1 = 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F

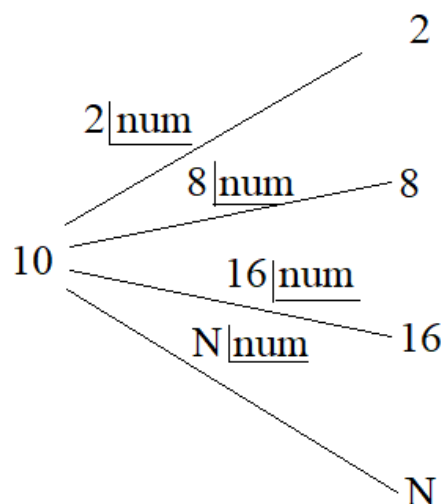
Base or Radix :

The value representing the system or maximum value of digits (or) symbols (or) letters used in the number system.

Number System Conversions:

1. Decimal to any other Number Systems.
2. Any other Number System to Decimal.
3. Octal to Binary Vs Binary to Octal.
4. HexaDecimal to Binary Vs Binary to Hexadecimal.
5. Octal to Hexadecimal Vs Hexadecimal to Octal.

1. Decimal to any other Number Systems:



Ex: 1. $(24)_{10}$ _____ $(\quad)_2$

$$\begin{array}{r|l}
 2 & 24 \\
 \hline
 2 & 12 - 0 \\
 \hline
 2 & 6 - 0 \\
 \hline
 2 & 3 - 0 \\
 \hline
 & 1 - 1
 \end{array}$$

$(24)_{10}$ _____ $(11000)_2$

Ex 2: $(24)_{10}$ _____ $(\quad)_8$

$$\begin{array}{r|l}
 8 & 24 \\
 \hline
 & 3 - 0
 \end{array}$$

$(24)_{10}$ _____ $(30)_8$

Ex 3: $(24)_{10}$ _____ $(\quad)_H$

$$\begin{array}{r|l}
 16 & 24 \\
 \hline
 & 1 - 8
 \end{array}$$

$(24)_{10}$ _____ $(18)_H$

Ex 4: $(24.25)_{10}$ _____ $(\quad)_2$

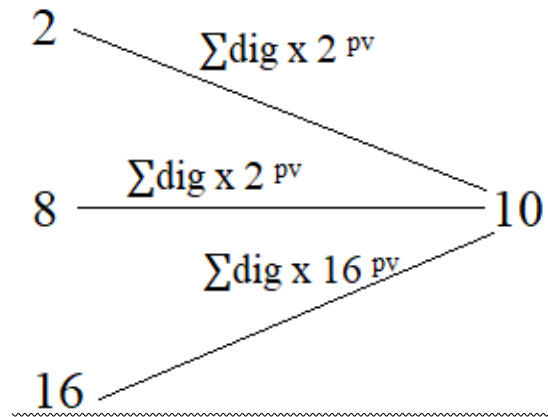
$$0.25 * 2 = 0.5 \quad \text{-- } 0$$

$$0.5 * 2 = 1.0 \quad \text{-- } 1$$

$$0.0 * 2 = 0.0$$

$(24.25)_{10}$ _____ $(11000.01)_2$

2. Any other to Decimal:



Ex 1:

$$(1101)_2 \text{ _____ } ()_{10}$$
$$0 \times 2^0 + 0 \times 2^1 + 1 \times 2^2 + 1 \times 2^3$$
$$0 + 0 + 4 + 8 = (12)_{10}$$

Ex 2:

$$(1100.01)_2 \text{ _____ } ()_{10}$$
$$1 \times 2^{-2} + 0 \times 2^{-1} + 0 \times 2^0 + 0 \times 2^1 + 1 \times 2^2 + 1 \times 2^3$$
$$0.25 + 0 + 0 + 0 + 4 + 8 = (12.25)_{10}$$

3. Octal to Binary Vs Binary to Octal:

Octal	Binary		
	2^2	2^1	2^0
0	0	0	0
1	0	0	1
2	0	1	0
3	0	1	1
4	1	0	0
5	1	0	1
6	1	1	0
7	1	1	1

Ex 1:

$(64)_8$ _____ $()_2$

6 4
↓ ↓

$(110.100)_2$

Ex 2:

$(64.25)_8$ _____ $()_2$

6 4 2 5
↓ ↓ ↓ ↓

$(110100.010101)_2$

Ex 3:

$(1010.11)_2$ ----- $()_8$

001 010 . 110
↓ ↓ ↓
1 2 6

$(12.6)_8$

4. Hexadecimal to Binary Vs Binary to Hexadecimal:

Hexadecimal	Binary			
	2^3	2^2	2^1	2^0
0	0	0	0	0
1	0	0	0	1
2	0	0	1	0
3	0	0	1	1
4	0	1	0	0
5	0	1	0	1
6	0	1	1	0
7	0	1	1	1
8	1	0	0	0
9	1	0	0	1
10 A	1	0	1	0
11 B	1	0	1	1
12 C	1	1	0	0
13 D	1	1	0	1
14 E	1	1	1	0
15 F	1	1	1	1

Ex 1 :

(64.25)_H ----- ()₂

6	4.	2	5
↓	↓	↓	↓

(0110 0100 . 0010 0101)₂

Ex 2 :

(10010 . 1101)₂ ----- ()_H

0001 0010 . 1101

(12D)_H

5. HexaDecimal to Octal and Octal to HexaDecimal Conversion :

* No direct conversion.

* But with the help of the intermediate i.e. (Binary and Decimal Conversions).

Ex 1:

(AF . 2B)_H ----- ()₈

Binary : 010 101 111 . 001 010 110

(257.126)₈

Ex 2 :

(12.1)₈ ----- ()_H

0011 1010.0010

(3A.2)_H

INTRODUCTION TO ALGORITHMS

Algorithm is a finite sequence of instructions, each of which has a clear meaning and can be performed with a finite amount of effort in a finite length of time. No matter what the input values may be, an algorithm terminates after executing a finite number of instructions.

We represent an algorithm using a pseudo language that is a combination of the constructs of a programming language together with informal English statements.

The ordered set of instructions required to solve a problem is known as an *algorithm*.

The characteristics of a good algorithm are:

- Precision – the steps are precisely stated (defined).
- Uniqueness – results of each step are uniquely defined and only depend on the input and the result of the preceding steps.
- Finiteness – the algorithm stops after a finite number of instructions are executed.
- Input – the algorithm receives input.
- Output – the algorithm produces output.
- Generality – the algorithm applies to a set of inputs.

Advantages of Algorithms:

1. It is a step-wise representation of a solution to a given problem, which makes it easy to understand.
2. An algorithm uses a definite procedure.
3. It is not dependent on any programming language, so it is easy to understand for anyone even without programming knowledge.
4. Every step in an algorithm has its own logical sequence so it is easy to debug.
5. By using algorithm, the problem is broken down into smaller pieces or steps hence; it is easier for programmer to convert it into an actual program.

Disdvantages of Algorithms:

1. Alogorithms is Time consuming.
2. Difficult to show Branching and Looping in Algorithms.
3. Big tasks are difficult to put in Algorithms.

Example

Write a algorithm to find out number is odd or even?

Ans.

step 1 : start

step 2 : input number

step 3 : rem=number mod 2

step 4 : if rem=0 then

 print "number even"

```
    else
        print "number odd"
    endif
step 5 : stop
```

practice:

write an algorithm to









1. Calculate the addition of two numbers
2. Compute and print quotient and remainder of two numbers A and B
3. Find sum and average of n numbers
4. Convert a decimal number to octal number
5. Print even numbers from 2 to 100
6. Print all even numbers and odd numbers between any given two integers N1 and N2($N1 < N2$) and also print the sum of all even and odd numbers.
7. Input the date in the dd/mm/yyyy format and print YES if it is a valid date otherwise print NO.
8. Sort given list of integers.
9. Find greatest of 3 numbers
10. Find a number is odd or even
11. Find a number is prime or not

FLOWCHART

Flowchart is a diagrammatic representation of an algorithm. Flowchart is very helpful in writing program and explaining program to others.

Symbols Used In Flowchart:

Different symbols are used for different states in flowchart, for example: Input/Output and decision making has different symbols. The table below describes all the symbols that are used in making flowchart

Symbol	Purpose	Description
	Flow line	Used to indicate the flow of logic by connecting symbols.
	Terminal(Stop/Start)	Used to represent start and end of flowchart.
	Input/Output	Used for input and output operation.
	Processing	Used for arithmetic operations and data-Manipulations.
	Decision	Used to represent the operation in which there are two alternatives, true and false.
	On-page Connector	Used to join different flow line
	Off-page Connector	Used to connect flowchart portion on different page.
	Predefined Process/Function	Used to represent a group of statements performing one processing task.

Advantages Of Using FLOWCHARTS:

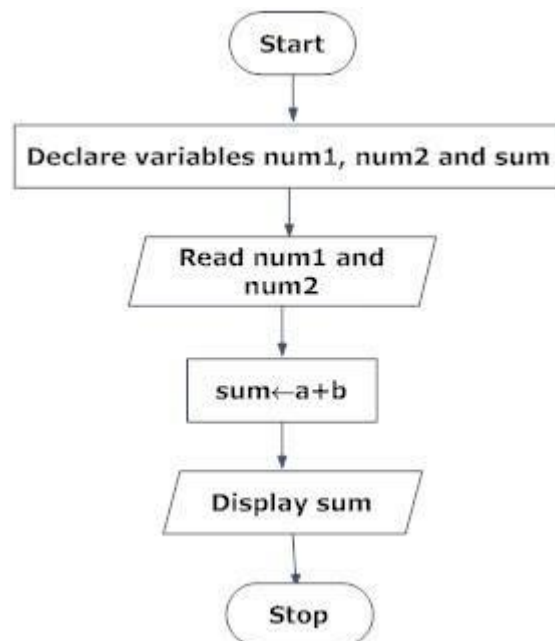
1. Communication: Flowcharts are better way of communicating the logic of a system to all concerned or involved.
2. Effective analysis: With the help of flowchart, problem can be analysed in more effective way therefore reducing cost and wastage of time.
3. Proper documentation: Program flowcharts serve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

Disadvantages Of Using FLOWCHARTS:

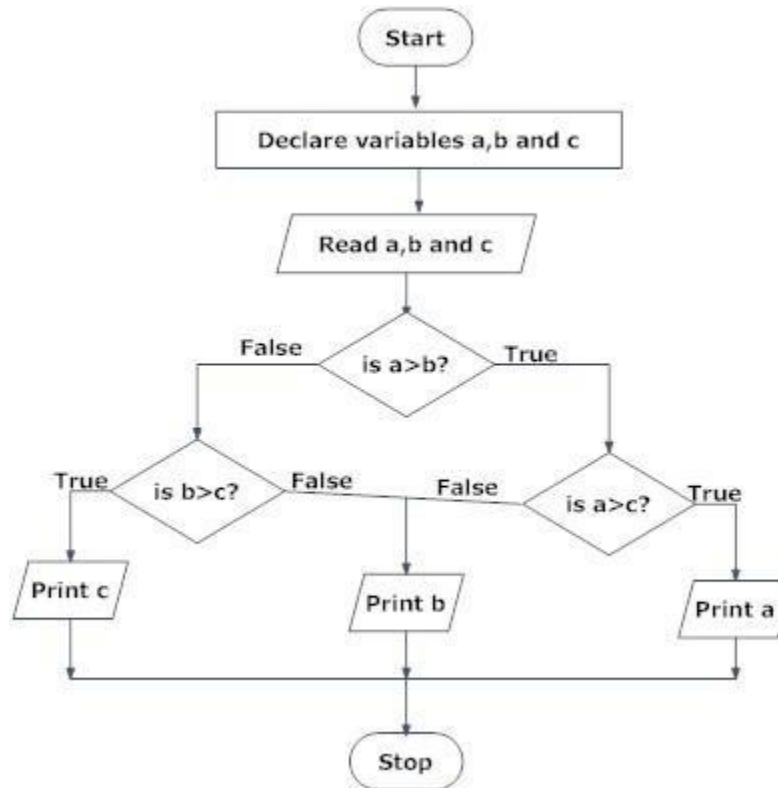
1. Complex logic: Sometimes, the program logic is quite complicated. In that case, flowchart becomes complex and clumsy. This will become a pain for the user, resulting in a waste of time and money trying to correct the problem
2. Alterations and Modifications: If alterations are required the flowchart may require re-drawing completely. This will usually waste valuable time.
3. Reproduction: As the flowchart symbols cannot be typed, reproduction of flowchart becomes a problem.

Examples of flowcharts in programming

Draw a flowchart to add two numbers entered by user.



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



Practice: 😊

write an algorithm and flowchart to

1. Swap the contents of two variables
2. Find largest of three numbers
3. Find sum and average of 3 numbers
4. Find whether the number is odd or even
5. Find whether a given number is prime or not
6. Find the sum of digits of a given number
7. Find the sum of digits of given number reduced to single digit
8. Reverse the digits in the given number and form a new number
9. Print first N fibonacci numbers.
10. Find factorial of a given number
11. Find the sum of array of elements
12. Calculate the number of words in a given sentence
13. Find the summation of all the integers between 10 and 500 which are divisible by 3 and 5 both. Also draw the flow chart for the same.
14. Arranging the numbers a, b and c in ascending order.
15. Print first N Fibonacci numbers.
16. Find the sum of first N natural numbers.

PSEUDO CODE:

Pseudocode is an informal high-level description of a computer program or algorithm. It is written in symbolic code which must be translated into a programming language before it can be executed.

Advantages of Pseudocode:

1. Improves the readability of any approach.
2. It's one of the best approaches to start implementation of an algorithm.
3. Acts as a bridge between the program and the algorithm or flowchart.
4. Also works as a rough documentation, so the program of one developer can be understood easily when a pseudo code is written out.
5. In industries, the approach of documentation is essential. And that's where a pseudo-code proves vital.
6. The main goal of a pseudo code is to explain what exactly each line of a program should do, hence making the code construction phase easier for the programmer.

Disadvantages of Pseudocode:

1. It doesn't provide visual representation of the program's logic
2. There are no accepted standards for writing pseudocodes. Designers use their own style of writing pseudocode.
3. Pseudo can not be compiled not executed hence its correctness cannot be verified by the computer.

In Pseudocode, they are used to indicate common input-output and processing operations. They are written fully in uppercase.

START: This is the start of your pseudocode.

INPUT: This is data retrieved from the user through typing or through an input device.

READ / GET: This is input used when reading data from a data file.

PRINT, DISPLAY, SHOW: This will show your output to a screen or the relevant output device.

COMPUTE, CALCULATE, DETERMINE: This is used to calculate the result of an expression.

SET, INIT: To initialize values

INCREMENT, BUMP: To increase the value of a variable

DECREMENT: To reduce the value of a variable

CONDITIONALS

During algorithm development, we need statements which evaluate expressions and execute instructions depending on whether the expression evaluated to True or False. Here are some common conditions used in Pseudocode:

IF — ELSE IF — ELSE

This is a conditional that is used to provide statements to be executed if a certain condition is met. This also applies to multiple conditions and different variables.

Here is an if statement with one condition

```
IF you are happy
  THEN smile
ENDIF
```

Here is an if statement with an else section. Else allows for some statements to be executed if the “if” condition is not met.

```
IF you are happy THEN
  smile
ELSE
  frown
ENDIF
```

We can add additional conditions to execute different statements if met.

```
IF you are happy THEN
  smile
ELSE IF you are sad
  frown
ELSE
  keep face plain
ENDIF
```

CASE

Case structures are used if we want to compare a single variable against several conditions.

```
INPUT color

CASE color of
```

```
red: PRINT "red"
green: PRINT "green"
blue: PRINT "blue"
```

OTHERS

```
PRINT "Please enter a value color"ENDCASE
```

The OTHERS clause with its statement is optional. Conditions are normally numbers or characters

ITERATION

To iterate is to repeat a set of instructions in order to generate a sequence of outcomes. We iterate so that we can achieve a certain goal.

FOR structure

The FOR loop takes a group of elements and runs the code within the loop for each element.

```
FOR every month in a year
```

```
    Compute number of days
```

```
ENDFOR
```

WHILE structure

Similar to the FOR loop, the while loop is a way to repeat a block of code as long as a predefined condition remains true. Unlike the FOR loop, the while loop evaluates based on how long the condition will remain true.

To avoid a scenario where our while loop runs infinitely, we add an operation to manipulate the value within each iteration. This can be through an increment, decrement, *et cetera*.

```
PRECONDITION: variable X is equal to 1
```

```
WHILE Population < Limit
```

```
    Compute Population as Population + Births — Deaths
```

```
ENDWHILE
```

FUNCTIONS

When solving advanced tasks it is necessary to break down the concepts in block of statements in different locations. This is especially true when the statements in question serve a particular purpose.

To reuse this code, we create functions. We can then call these functions every-time we need them to run.

Function clear monitor

Pass In: nothing

Direct the operating system to clear the monitor

Pass Out: nothing

Endfunction

To emulate a function call in pseudocode, we can use the **Call** keyword
call: clear monitor

practice:

Write a pseudocode to

1. Read number n and print the integers counting upto n
2. Read number n and print the sum of the integers upto n.
3. Add two matrices
4. Solve quadratic equation

INTRODUCTION TO C LANGUAGE

C is a general-purpose high level language that was originally developed by **Dennis Ritchie** for the UNIX operating system. It was first implemented on the Digital Equipment Corporation PDP-11 computer in **1972**.

The UNIX operating system and virtually all UNIX applications are written in the C language. C has now become a widely used professional language for various reasons.

- Easy to learn
- Structured language
- It produces efficient programs.
- It can handle low-level activities.
- It can be compiled on a variety of computers.

Facts about C

- C was invented to write an operating system called UNIX.
- C is a successor of B language which was introduced around 1970
- The language was formalized in 1988 by the American National Standard Institute (ANSI).
- By 1973 UNIX OS almost totally written in C.
- Today C is the most widely used System Programming Language.
- Most of the state of the art software have been implemented using C

Why to use C?

C was initially used for system development work, in particular the programs that make-up the operating system. C was adopted as a system development language because it produces code that runs nearly as fast as code written in assembly language. Some examples of the use of C might be:

- Operating Systems
- Language Compilers
- Assemblers
- Text Editors
- Print Spoolers
- Network Drivers

- Modern Programs
- Data Bases
- Language Interpreters
- Utilities

C Program File

All the C programs are written into text files with extension ".c" for example *hello.c*. You can use "vi" editor to write your C program into a file.

HISTORY OF C LANGUAGE

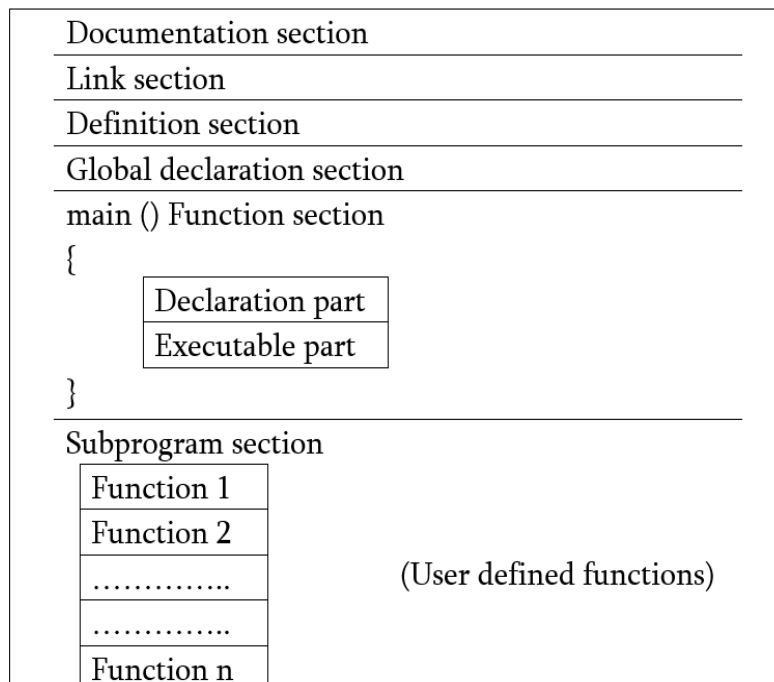
C is a general-purpose language which has been closely associated with the **UNIX** operating system for which it was developed - since the system and most of the programs that run it are written in C.

Many of the important ideas of C stem from the language **BCPL**, developed by Martin Richards. The influence of BCPL on C proceeded indirectly through the language **B**, which was written by Ken Thompson in 1970 at Bell Labs, for the first UNIX system on a **DEC** PDP-**BCPL** and **B** are "type less" languages whereas C provides a variety of data types.

In 1972 Dennis Ritchie at Bell Labs writes C and in 1978 the publication of The C Programming Language by Kernighan & Ritchie caused a revolution in the computing world.

In 1983, the American National Standards Institute (ANSI) established a committee to provide a modern, comprehensive definition of C. The resulting definition, the ANSI standard, or "ANSI C", was completed late 1988.

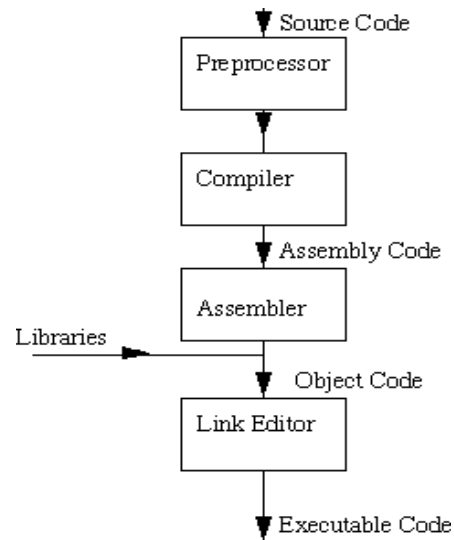
BASIC STRUCTURE OF C PROGRAMMING



1. **Documentation section:** The documentation section consists of a set of comment lines giving the name of the program, the author and other details, which the programmer would like to use later.
2. **Link section:** The link section provides instructions to the compiler to link functions from the system library such as using the #include directive.
3. **Definition section:** The definition section defines all symbolic constants such using the #define directive.
4. **Global declaration section:** There are some variables that are used in more than one function. Such variables are called global variables and are declared in the global declaration section that is outside of all the functions. This section also declares all the user-defined functions.
5. **main() function section:** Every C program must have one main function section. This section contains two parts; declaration part and executable part
 1. **Declaration part:** The declaration part declares all the variables used in the executable part.
 2. **Executable part:** There is at least one statement in the executable part. These two parts must appear between the opening and closing braces. The program execution begins at the opening brace and ends at the closing brace. The closing brace of the main function is the logical end of the program. All statements in the declaration and executable part end with a semicolon.
6. **Subprogram section:** If the program is a multi-function program then the subprogram section contains all the user-defined functions that are called in the main () function. User-defined functions are generally placed immediately after the main () function, although they may appear in any order.

PROCESS OF COMPILING AND RUNNING C PROGRAM

We will briefly highlight key features of the C Compilation model here.



The C Compilation Model

The Preprocessor

The Preprocessor accepts source code as input and is responsible for

- removing comments
- Interpreting special *preprocessor directives* denoted by #.

For example

- `#include` -- includes contents of a named file. Files usually called *header* files. *e.g*
 - `#include <math.h>` -- standard library maths file.
 - `#include <stdio.h>` -- standard library I/O file
- `#define` -- defines a symbolic name or constant. Macro substitution.
 - `#define MAX_ARRAY_SIZE 100`

C Compiler

The C compiler translates source to assembly code. The source code is received from the preprocessor.

Assembler

The assembler creates object code. On a UNIX system you may see files with a .o suffix (.OBJ on MSDOS) to indicate object code files.

Link Editor

If a source file references library functions or functions defined in other source files the *link editor* combines these functions (with main()) to create an executable file.

C TOKENS

C tokens are the basic building blocks in C language which are constructed together to write a C program.

Each and every smallest individual unit in a C program is known as C tokens.

C tokens are of six types. They are

Keywords	(eg: int, while),
Identifiers	(eg: main, total),
Constants	(eg: 10, 20),
Strings	(eg: "total", "hello"),
Special symbols	(eg: (), {}),
Operators	(eg: +, /, -, *)

C KEYWORDS

C keywords are the words that convey a special meaning to the c compiler. The keywords cannot be used as variable names.

The list of C keywords is given below:

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

C IDENTIFIERS

Identifiers are used as the general terminology for the names of variables, functions and arrays. These are user defined names consisting of arbitrarily long sequence of letters and digits with either a letter or the underscore (_) as a first character.

There are certain rules that should be followed while naming c identifiers:

They must begin with a letter or underscore (_).

They must consist of only letters, digits, or underscore. No other special character is allowed.

It should not be a keyword.

It must not contain white space.

It should be up to 31 characters long as only first 31 characters are significant.

Some examples of c identifiers:

Name	Remark
_A9	Valid
Temp.var	Invalid as it contains special character other than the underscore
void	Invalid as it is a keyword

C CONSTANTS

A C constant refers to the data items that do not change their value during the program execution. Several types of C constants that are allowed in C are:

Integer Constants

Integer constants are whole numbers without any fractional part. It must have at least one digit and may contain either + or – sign. A number with no sign is assumed to be positive.

There are three types of integer constants:

Decimal Integer Constants

Integer constants consisting of a set of digits, 0 through 9, preceded by an optional – or + sign.

Example of valid decimal integer constants

341, -341, 0, 8972

Octal Integer Constants

Integer constants consisting of sequence of digits from the set 0 through 7 starting with 0 is said to be octal integer constants.

Example of valid octal integer constants

010, 0424, 0, 0540

Hexadecimal Integer Constants

Hexadecimal integer constants are integer constants having sequence of digits preceded by 0x or

0X. They may also include alphabets from A to F representing numbers 10 to 15.

Example of valid hexadecimal integer constants

0xD, 0X8d, 0X, 0xbD

It should be noted that, octal and hexadecimal integer constants are rarely used in programming.

Real Constants

The numbers having fractional parts are called real or floating point constants. These may be represented in one of the two forms called *fractional form* or the *exponent form* and may also have either + or – sign preceding it.

Example of valid real constants in fractional form or decimal notation

0.05, -0.905, 562.05, 0.015

Representing a real constant in exponent form

The general format in which a real number may be represented in exponential or scientific form is

mantissa e exponent

The mantissa must be either an integer or a real number expressed in decimal notation.

The letter e separating the mantissa and the exponent can also be written in uppercase i.e. E. And, the exponent must be an integer.

Examples of valid real constants in exponent form are:

252E85, 0.15E-10, -3e+8

Character Constants

A character constant contains one single character enclosed within single quotes.

Examples of valid character constants

‘a’, ‘Z’, ‘5’

It should be noted that character constants have numerical values known as ASCII values, for example, the value of ‘A’ is 65 which is its ASCII value.

Escape Characters/ Escape Sequences

C allows us to have certain non graphic characters in character constants. Non graphic characters are those characters that cannot be typed directly from keyboard, for example, tabs, carriage return, etc.

These non graphic characters can be represented by using escape sequences represented by a backslash() followed by one or more characters.

NOTE: An escape sequence consumes only one byte of space as it represents a single character.

Escape Sequence	Description
a	Audible alert(bell)
b	Backspace
f	Form feed
n	New line
r	Carriage return
t	Horizontal tab
v	Vertical tab
\	Backslash
“	Double quotation mark
‘	Single quotation mark
?	Question mark
	Null

STRING CONSTANTS

String constants are sequence of characters enclosed within double quotes. For example, “hello”

“abc”

“hello911”

Every sting constant is automatically terminated with a special character ‘\0’ called the **null character** which represents the end of the string.

For example, “hello” will represent “hello” in the memory.

Thus, the size of the string is the total number of characters plus one for the null character.

Special Symbols

The following special symbols are used in C having some special meaning and thus, cannot be used for some other purpose. [] () {} , ; : * ... = #

Braces{}: These opening and ending curly braces marks the start and end of a block of code containing more than one executable statement.

Parentheses(): These special symbols are used to indicate function calls and function parameters.

Brackets[]: Opening and closing brackets are used as array element reference. These indicate single and multidimensional subscripts.

DATA TYPES IN C:

Data types in C refer to an extensive system used for declaring variables or functions of different types. The type of a variable determines how much space it occupies in storage and how the bit pattern stored is interpreted.

The types in C can be classified as follows:

S.N.	Types and Description
1	Basic Types: They are arithmetic types and are further classified into: (a) integer types and (b) floating-point types.
2	Enumerated types: They are again arithmetic types and they are used to define variables that can only assign certain discrete integer values throughout the program.
3	The type void: The type specifier <i>void</i> indicates that no value is available.
4	Derived types: They include (a) Pointer types, (b) Array types, (c) Structure types, (d) Union types, and (e) Function types.

Integer Types

The following table provides the details of standard integer types with their storage sizes and value ranges:

Type	Storage size	Value range
char	1 byte	-128 to 127 or 0 to 255
unsigned char	1 byte	0 to 255

signed char	1 byte	-128 to 127
int	2 or 4 bytes	-32,768 to 32,767 or -2,147,483,648 to 2,147,483,647
unsigned int	2 or 4 bytes	0 to 65,535 or 0 to 4,294,967,295
short	2 bytes	-32,768 to 32,767
unsigned short	2 bytes	0 to 65,535
long	4 bytes	-2,147,483,648 to 2,147,483,647
unsigned long	4 bytes	0 to 4,294,967,295

To get the exact size of a type or a variable on a particular platform, you can use the **sizeof** operator. The expressions `sizeof(type)` yields the storage size of the object or type in bytes. Given below is an example to get the size of int type on any machine:

```
#include <stdio.h>
#include <limits.h>
int main()
{
    printf("Storage size for int : %d \n", sizeof(int));
    return 0;
}
```

When you compile and execute the above program, it produces the following result on Linux:

```
Storage size for int : 4
```

Floating-Point Types

The following table provides the details of standard floating-point types with storage sizes and value ranges and their precision:

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

The header file `float.h` defines macros that allow you to use these values and other details about the binary representation of real numbers in your programs. The following example prints the storage space taken by a float type and its range values:

```
#include <stdio.h>
#include <float.h>
int main()
{
    printf("Storage size for float : %d \n", sizeof(float));
    printf("Minimum float positive value: %E\n", FLT_MIN );
    printf("Maximum float positive value: %E\n", FLT_MAX );
    printf("Precision value: %d\n", FLT_DIG );
    return 0;
}
```

When you compile and execute the above program, it produces the following result on Linux:

Storage size for float : 4

Minimum float positive value: 1.175494E-38

Maximum float positive value: 3.402823E+38

Precision value: 6

The void Type

The void type specifies that no value is available. It is used in three kinds of situations:

S.N.	Types and Description
1	Function returns as void There are various functions in C which do not return any value or you can say they return void. A function with no return value has the return type as void. For example, void exit (int status);
2	Function arguments as void There are various functions in C which do not accept any parameter. A function with no parameter can accept a void. For example, int rand(void);
3	Pointers to void A pointer of type void * represents the address of an object, but not its type. For example, a memory allocation function void *malloc(size_t size); returns a pointer to void which can be casted to any data type.

VARIABLES

A variable is nothing but a name given to a storage area that our programs can manipulate. Each variable in C has a specific type, which determines the size and layout of the variable's memory; the range of values that can be stored within that memory; and the set of operations that can be applied to the variable.

The name of a variable can be composed of letters, digits, and the underscore character. It must begin with either a letter or an underscore. Upper and lowercase letters are distinct because C is case-sensitive. Based on the basic types explained in the previous chapter, there will be the following basic variable types –

Type	Description
char	Typically a single octet(one byte). This is an integer type.
int	The most natural size of integer for the machine.
float	A single-precision floating point value.
double	A double-precision floating point value.
void	Represents the absence of type.

C programming language also allows defining various other types of variables like Enumeration, Pointer, Array, Structure, Union, etc.

Variable Definition in C

A variable definition tells the compiler where and how much storage to create for the variable. A variable definition specifies a data type and contains a list of one or more variables of that type as follows –

```
type variable_list;
```

Here, **type** must be a valid C data type including char, w_char, int, float, double, any user-defined object; and **variable_list** may consist of one or more identifier names separated by commas. Some valid declarations are shown here –

```
int i, j, k;
char c, ch;
float f, salary;
double d;
```

The line **int i, j, k;** declares and defines the variables i, j, and k; which instruct the compiler to create variables named i, j and k of type int.

Variables can be initialized (assigned an initial value) in their declaration. The initializer consists of an equal sign followed by a constant expression as follows –

```
type variable_name = value;
```


Some examples are –

```
int d, f;           // declaration of d and f.  
int d = 3, f = 5;   // definition and initializing d and f.  
char x = 'x';       // the variable x has the value 'x'.
```

For definition without an initializer: variables with static storage duration are implicitly initialized with NULL (all bytes have the value 0); the initial value of all other variables are undefined.

Variable Declaration in C:

A variable declaration provides assurance to the compiler that there exists a variable with the given type and name so that the compiler can proceed for further compilation without requiring the complete detail about the variable. A variable definition has its meaning at the time of compilation only; the compiler needs actual variable definition at the time of linking the program.

OPERATORS AND EXPRESSIONS

C language offers many types of operators. They are,

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

S.no	Types of Operators	Description
1	Arithmetic_operators	These are used to perform mathematical calculations like addition, subtraction, multiplication, division and modulus
2	Assignment_operators	These are used to assign the values for the variables in C programs.
3	Relational operators	These operators are used to compare the value of two variables.
4	Logical operators	These operators are used to perform logical operations on the given two variables.
5	Bit wise operators	These operators are used to perform bit operations on given two variables.
6	Conditional (ternary) operators	Conditional operators return one value if condition is true and returns another value if condition is false.
7	Increment/decrement operators	These operators are used to either increase or decrease the value of the variable by one.
8	Special operators	&, *, sizeof() and ternary operators.

ARITHMETIC OPERATORS IN C

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

S.no	Arithmetic Operators	Operation	Example
1	+	Addition	A+B
2	−	Subtraction	A-B
3	*	multiplication	A*B
4	/	Division	A/B
5	%	Modulus	A%B

EXAMPLE PROGRAM FOR C ARITHMETIC OPERATORS

In this example program, two values “40” and “20” are used to perform arithmetic operations such as addition, subtraction, multiplication, division, modulus and output is displayed for each operation.

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

Addition of a, b is : 60

Subtraction of a, b is : 20

Multiplication of a, b is : 800

Division of a, b is : 2

Modulus of a, b is : 0

ASSIGNMENT OPERATORS IN C

In C programs, values for the variables are assigned using assignment operators.

For example, if the value “10” is to be assigned for the variable “sum”, it can be assigned as “sum = 10;”

Other assignment operators in C language are given below.

Operators		Example	Explanation
Simple assignment operator	=	sum = 10	10 is assigned to variable sum
Compound assignment operators	+=	sum += 10	This is same as sum = sum + 10
	-=	sum -= 10	This is same as sum = sum – 10
	*=	sum *= 10	This is same as sum = sum * 10
	/=	sum /= 10	This is same as sum = sum / 10
	%=	sum %= 10	This is same as sum = sum % 10
	&=	sum&=10	This is same as sum = sum & 10
	^=	sum ^= 10	This is same as sum = sum ^ 10

```
# include <stdio.h>
int main()
{
int Total=10;
Total+=2; // This is same as Total = Total+2
printf("Total = %d", Total);
}
```

Total = 12

RELATIONAL OPERATORS IN C

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

S.no	Operators	Example	Description
1	>	x > y	x is greater than y
2	<	x < y	x is less than y
3	>=	x >= y	x is greater than or equal to y
4	<=	x <= y	x is less than or equal to y
5	==	x == y	x is equal to y
6	!=	x != y	x is not equal to y

These operators returns True (non-zero) if the condition is satisfied, otherwise False (zero).

EXAMPLE PROGRAM FOR RELATIONAL OPERATORS IN C

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

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

```
#include <stdio.h>
int main()
{
int m=40,n=20;
printf("m and n are equal (m==n)==>%d",(m==n));
}
```

m and n are not equal (m==n)==>0

LOGICAL OPERATORS IN C

These operators are used to perform logical operations on the given expressions. we can combine two relational expressions with logical operators

There are 3 logical operators in C language. They are, logical AND (&&), logical OR (||) and logical NOT (!).

S.no	Operators	Name	Example	Description
1	&&	logical AND	(x>5)&&(y<5)	It returns true when both conditions are true
2		logical OR	(x>=10) (y>=10)	It returns true when at-least one of the condition is true
3	!	logical NOT	!((x>5)&&(y<5))	It reverses the state of the operand “((x>5) &&(y<5))” If “((x>5)&& (y<5))” is true, logical NOT operator makes it false

PROGRAM ON LOGICAL OPERATORS

```
#include <stdio.h>
int main()
{
int m=40,n=20;
printf("(m==n)&&(m!=n)==>%d",(m==n)&&(m!=n));
printf("(m==n)|| (m!=n)==>%d",(m==n)|| (m!=n));
printf("! (m==30)==>%d",! (m==30));
}
```

```
(m==n)&&(m!=n)==>0
```

```
(m==n)||(m!=n)==>1
```

```
!(m==40)=0
```

&& operator – it returns true only when both conditions (m==n and m!=n) is true. Else, it becomes false.

|| Operator – it returns true when any one of the condition (m==n || m!=n) is true. It becomes false when none of the condition is true.

! Operator – It is used to reverses the state of the operand.

If the conditions (m==40) is true, true (1) is returned. This value is inverted by “!” operator.

So, “!(m==40)” returns false (0).

BIT WISE OPERATORS IN C

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

Bit wise operators in C language are & (bitwise AND), | (bitwise OR), ~ (bitwise NOT), ^ (XOR),

<< (left shift) and >> (right shift).

Operator_symbol	Operator_name
&	Bitwise_AND
	Bitwise OR
~	Bitwise_NOT
^	XOR
<<	Left Shift
>>	Right Shift

TRUH TABLE FOR BIT WISE OPERATION BIT WISE OPERATORS

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

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

x = 00101000 y = 01010000

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

$x \& y = 00000000$ (binary) = 0 (decimal)

$$x|y = 01111000 \text{ (binary)} = 120 \text{ (decimal)}$$
$$\mathbf{z}_X = \text{11010111}$$

.. ..= -41 (decimal)

$$x^y = 01111000 \text{ (binary)} = 120 \text{ (decimal)}$$

$x \ll 1 = 01010000$ (binary) = 80 (decimal)

$x \gg 1 = 00010100$ (binary) = 20 (decimal)

Note:

Bitwise NOT:

[illegible]

So, all 0's are converted into 1's in bit wise NOT operation.

Bit wise left shift and right shift : In left shift operation “ $x \ll 1$ “, 1 means that the bits will be left shifted by one place. If we use it as “ $x \ll 2$ “, then, it means that the bits will be left shifted by 2 places.

EXAMPLE PROGRAM FOR BIT WISE OPERATORS IN C

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

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

int m = 40,n = 80,AND_opr,OR_opr,XOR_opr,NOT_opr ;
AND_opr = (m&n);
OR_opr   =   (m|n);
```



```

NOT_opr    =    (~m);
XOR_opr = (m^n);
printf("AND_opr value = %d\n",AND_opr );
printf("OR_opr value = %d\n",OR_opr );
printf("NOT_opr value = %d\n",NOT_opr );

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

```

OUTPUT:

```

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

```

CONDITIONAL OR TERNARY OPERATORS IN C

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

Syntax : (Condition? true_value: false_value);

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

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

EXAMPLE PROGRAM FOR CONDITIONAL/TERNARY OPERATORS IN C

```

#include <stdio.h>
int main()
{
int x=1, y ;

```

```
y = ( x ==1 ? 2 : 0 );  
printf("x value is %d\n", x);  
printf("y value is %d", y);  
}
```

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

C – Increment/decrement Operators

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

Syntax:

Increment operator: ++var_name ;(or) var_name++;

Decrement operator: --var_name; (or) var_name --;

Example:

Increment operator : ++ i ; i ++ ;

Decrement operator : -- i ; i -- ;

EXAMPLE PROGRAM FOR INCREMENT OPERATORS IN C

In this program value of “i” is incremented one using “i++” operator and output is displayed as 2.

```
//Example for increment operators  
#include <stdio.h>  
int main()  
{  
int i=1;  
i++;  
printf(“%d”,i);  
}
```

```
2
```

EXAMPLE PROGRAM FOR DECREMENT OPERATORS IN C

In this program value of “i” is decremented one using “i--” operator and output is displayed as 4.

```
//Example for decrement operators //Example for increment operators
#include <stdio.h>
int main()
{
int i=5;
i--;
printf(“%d”,i);
}
```

4

DIFFERENCE BETWEEN PRE/POST INCREMENT & DECREMENT OPERATORS IN C

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

S.no	Operator type	Operator	Description
1	Pre increment	++i	Value of i is incremented before assigning it to variable i.
2	Post-increment	i++	Value of i is incremented after assigning it to variable i.
3	Pre decrement	--i	Value of i is decremented before assigning it to variable i.
4	Post_decrement	i--	Value of i is decremented after assigning it to variable i.

```
/* Increment and Decrement Operators in C as Prefix and Postfix */
```

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

```
int main()  
{
```

```
int x = 10,y = 20, a = 5, b= 4;
```

```
printf("---- PRE INCREMENT OPERATOR EXAMPLE---- \n");  
printf("Value of x : %d \n", x); //Original Value  
printf("Value of x : %d \n", ++x); // using Pre increment Operator  
printf("Value of Incremented x : %d \n", x); //Incremented value
```

```
printf("----POST INCREMENT OPERATOR EXAMPLE---- \n");  
printf("Value of y : %d \n", y); //Original Value  
printf("Value of y : %d \n", y++); // using Post increment Operator  
printf("Value of Incremented y : %d \n", y); //Incremented value
```

```
printf("----PRE DECREMENT OPERATOR EXAMPLE---- \n");  
printf("Value of a : %d \n", a); //Original Value  
printf("Value of a : %d \n", --a); // using Pre decrement Operator  
printf("Value of decremented y : %d \n", a); //decremented value
```

```
printf("----POST DECREMENT OPERATOR EXAMPLE---- \n");  
printf("Value of b : %d \n", b); //Original Value  
printf("Value of b : %d \n", b--); // using Post decrement Operator  
printf("Value of decremented b : %d \n", b); //decremented value
```

```
return 0;  
}
```

```
---- PRE INCREMENT OPERATOR EXAMPLE----
```

```
Value of x : 10
```

```
Value of x : 11
```

```
Value of Incremented x : 11
```

```
---- POST INCREMENT OPERATOR EXAMPLE----
```

```
Value of y : 20
```

```
Value of y : 20
```

```
Value of Incremented y : 21
```

```
---- PRE DECREMENT OPERATOR EXAMPLE----
```

```
Value of a : 5
```

```
Value of a : 4
```

```
Value of decremented a : 4
```

```
---- POST DECREMENT OPERATOR EXAMPLE----
```

```
Value of b : 4
```

```
Value of b : 4
```

```
Value of decremented b : 3
```

SPECIAL OPERATORS IN C :

S.no	Operators	Description
1	&	This is used to get the address of the variable. Example : &a will give address of a.
2	*	This is used as pointer to a variable. Example : * a where, * is pointer to the variable a.
3	Sizeof ()	This gives the size of the variable. Example : size of (char) will give us 1.

EXAMPLE PROGRAM FOR & AND * OPERATORS IN C

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

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

    int *ptr, q;
    q = 50;
    /* address of q is assigned to ptr */
    ptr = &q;
    /* display q's value using ptr variable */
    printf("%d", *ptr);
    return 0;

}
```

50

EXAMPLE PROGRAM FOR sizeof () OPERATOR IN C

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

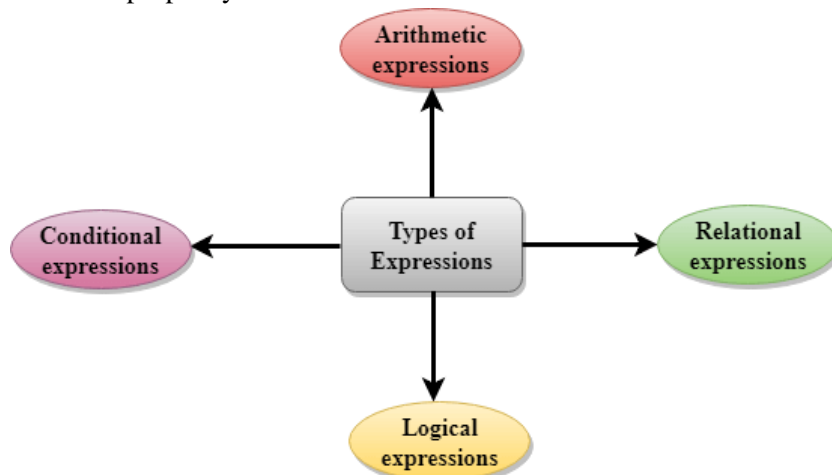
```
#include <stdio.h>
#include <limits.h>
int main()
{
    int a;
    char b;

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

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

EXPRESSIONS

Arithmetic expression in C is a combination of variables, constants and operators written in a proper syntax. C can easily handle any complex mathematical expressions but these mathematical expressions have to be written in a proper syntax. Some examples of expressions written in proper syntax of C are



Note: C does not have any operator for exponentiation.

Arithmetic Expressions

An arithmetic expression is an expression that consists of operands and arithmetic operators. An arithmetic expression computes a value of type int, float or double.

$6*2/(2+1 * 2/3 +6) +8 * (8/4)$

Relational Expressions

- A relational expression is an expression used to compare two operands.
- It is a condition which is used to decide whether the action should be taken or not.
- In relational expressions, a numeric value cannot be compared with the string value.
- The result of the relational expression can be either zero or non-zero value. Here, the zero value is equivalent to a false and non-zero value is equivalent to true.

$x \% 2 == 0$	This condition is used to check whether the x is an even number or not. The relational expression results in value 1 if x is an even number otherwise results in value 0.
---------------	---

Logical Expressions

- A logical expression is an expression that computes either a zero or non-zero value.
- It is a complex test condition to take a decision.

$(x > 4) \ \&\& \ (x < 6)$	It is a test condition to check whether the x is greater than 4 and x is less than 6. The result of the condition is true only when both the conditions are true.
----------------------------	---

Conditional Expressions

- A conditional expression is an expression that returns 1 if the condition is true otherwise 0.
- A conditional operator is also known as a ternary operator.

The Syntax of Conditional operator

Suppose exp1, exp2 and exp3 are three expressions.

$\text{exp1} ? \text{exp2} : \text{exp3}$

The above expression is a conditional expression which is evaluated on the basis of the value of the exp1 expression. If the condition of the expression exp1 holds true, then the final conditional expression is represented by exp2 otherwise represented by exp3.

C OPERATOR PRECEDENCE AND ASSOCIATIVITY

C operators in order of *precedence* (highest to lowest). Their associativity indicates in what order operators of equal precedence in an expression are applied.

Operator	Description	Associativity
() [] . -> ++ --	Parentheses (function call) (see Note 1) Brackets (array subscript) Member selection via object name Member selection via pointer Postfix increment/decrement (see Note 2)	left-to-right
++ -- + - ! ~ (type) * & sizeof	Prefix increment/decrement Unary plus/minus Logical negation/bitwise complement Cast (convert value to temporary value of <i>type</i>) Dereference Address (of operand) Determine size in bytes on this implementation	right-to-left
* / %	Multiplication/division/modulus	left-to-right
+ -	Addition/subtraction	left-to-right
<< >>	Bitwise shift left, Bitwise shift right	left-to-right
< <= > >=	Relational less than/less than or equal to Relational greater than/greater than or equal to	left-to-right
== !=	Relational is equal to/is not equal to	left-to-right
&	Bitwise AND	left-to-right
^	Bitwise exclusive OR	left-to-right
	Bitwise inclusive OR	left-to-right
&&	Logical AND	left-to-right
	Logical OR	left-to-right
? :	Ternary conditional	right-to-left
= += -= *= /= %= &= ^= = <<= >>=	Assignment Addition/subtraction assignment Multiplication/division assignment Modulus/bitwise AND assignment Bitwise exclusive/inclusive OR assignment Bitwise shift left/right assignment	right-to-left
,	Comma (separate expressions)	left-to-right

Note 1:

Parentheses are also used to group sub-expressions to force a different precedence; such parenthetical expressions can be nested and are evaluated from inner to outer.

Note 2:

Postfix increment/decrement have high precedence, but the actual increment or decrement of the operand is delayed (to be accomplished sometime before the statement completes execution). So in the statement $y = x * z++$; the current value of z is used to evaluate the expression (*i.e.*, $z++$ evaluates to z) and z only incremented after all else is done.

EVALUATION OF EXPRESSION

At first, the expressions within parenthesis are evaluated. If no parenthesis is present, then the arithmetic expression is evaluated from left to right. There are two priority levels of operators in C.

High priority: $*$ / $\%$

Low priority: $+$ $-$

The evaluation procedure of an arithmetic expression includes two left to right passes through the entire expression. In the first pass, the high priority operators are applied as they are encountered and in the second pass, low priority operations are applied as they are encountered. Suppose, we have an arithmetic expression as:

$x = 9 - 12 / 3 + 3 * 2 - 1$

This expression is evaluated in two left to right passes as:

First Pass

Step 1: $x = 9 - 4 + 3 * 2 - 1$

Step 2: $x = 9 - 4 + 6 - 1$

Second Pass

Step 1: $x = 5 + 6 - 1$

Step 2: $x = 11 - 1$

Step 3: $x = 10$

But when parenthesis is used in the same expression, the order of evaluation gets changed.

For example,

$x = 9 - 12 / (3 + 3) * (2 - 1)$

When parentheses are present then the expression inside the parenthesis are evaluated first from

left to right. The expression is now evaluated in three passes as:

First Pass

Step 1: $x = 9 - 12 / 6 * (2 - 1)$

Step 2: $x = 9 - 12 / 6 * 1$

Second Pass

Step 1: $x = 9 - 2 * 1$

Step 2: $x = 9 - 2$

Third Pass

Step 3: $x = 7$

There may even arise a case where nested parentheses are present (i.e. parenthesis inside parenthesis). In such case, the expression inside the innermost set of parentheses is evaluated first and then the outer parentheses are evaluated.

For example, we have an expression as:

$x = 9 - ((12 / 3) + 3 * 2) - 1$

The expression is now evaluated as:

First Pass:

Step 1: $x = 9 - (4 + 3 * 2) - 1$

Step 2: $x = 9 - (4 + 6) - 1$

Step 3: $x = 9 - 10 - 1$

Second Pass

Step 1: $x = -1 - 1$

Step 2: $x = -2$

Note: The number of evaluation steps is equal to the number of operators in the arithmetic expression.

TYPE CONVERSION IN EXPRESSIONS

When variables and constants of different types are combined in an expression then they are converted to same data type. The process of converting one predefined type into another is called type conversion.

Type conversion in c can be classified into the following two types:

Implicit Type Conversion

When the type conversion is performed automatically by the compiler without programmer's intervention, such type of conversion is known as **implicit type conversion** or **type promotion**. The compiler converts all operands into the data type of the largest operand.

The sequence of rules that are applied while evaluating expressions are given below:

All short and char are automatically converted to int, then,

If either of the operand is of type long double, then others will be converted to long double and result will be long double.

Else, if either of the operand is double, then others are converted to double.

Else, if either of the operand is float, then others are converted to float.

Else, if either of the operand is unsigned long int, then others will be converted to unsigned long int.

Else, if one of the operand is long int, and the other is unsigned int, then

if a long int can represent all values of an unsigned int, the unsigned int is converted to long int. otherwise, both operands are converted to unsigned long int.

Else, if either operand is long int then other will be converted to long int.

Else, if either operand is unsigned int then others will be converted to unsigned int.

It should be noted that the final result of expression is converted to type of variable on left side of assignment operator before assigning value to it.

Also, conversion of float to int causes truncation of fractional part, conversion of double to float causes rounding of digits and the conversion of long int to int causes dropping of excess higher order bits.

Explicit Type Conversion

The type conversion performed by the programmer by posing the data type of the expression of specific type is known as explicit type conversion.

The explicit type conversion is also known as **type casting**. Type casting in c is done in the following form:

(data_type)expression;

where, *data_type* is any valid c data type, and *expression* may be constant, variable or expression

For example, `x=(int)a+b*d;`

The following rules have to be followed while converting the expression from one type to another to avoid the loss of information:

All integer types to be converted to float.

All float types to be converted to double.

All character types to be converted to integer.

FORMATTED INPUT AND OUTPUT ☺

C provides standard functions `scanf()` and `printf()`, for performing formatted input and output.

These functions accept, as parameters, a format specification string and a list of variables.

The format specification string is a character string that specifies the data type of each variable to be input or output and the size or width of the input and output.

Now to discuss formatted output in functions.

Formatted Output

The function `printf()` is used for formatted output to standard output based on a format specification. The format specification string, along with the data to be output, are the parameters to the `printf()` function.

Syntax:

```
printf (format, data1, data2,.....);
```

In this syntax `format` is the format specification string. This string contains, for each variable to be output, a specification beginning with the symbol `%` followed by a character called the conversion character.

Example:

```
printf ("%c", data1);
```

The character specified after `%` is called a conversion character because it allows one data type to be converted to another type and printed.

See the following table conversion character and their meanings.

Conversion Character/ Control String	Meaning
d	The data is converted to decimal (integer)
c	The data is taken as a character.
s	The data is a string and character from the string are printed until a NULL character is reached.
f	The data is output as float or double with a default Precision 6.
Symbols	Meaning
\n	For new line (linefeed return)
\t	For tab space (equivalent of 8 spaces)

Example

```
printf ("%c\n",data1);
```

The format specification string may also have text.

Example

```
printf ("Character is:""%c\n", data1);
```

The text "Character is:" is printed out along with the value of data1.

Example with program

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

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

```
Main()
```

```
{
```

```
    Char alphabh="A";
```

```
    int number1=55;
```

```
    float number2=22.34;
```

```
    printf ("char=%c\n",alphabh);
```

```
    printf ("int=%d\n",number1);
```

```
    printf ("float=%f\n",number2);
```

```
    getch();
```

```
    clrscr();
```

```
    retrun 0;
```

```
}
```

```
char =A
```

```
int=55
```

```
flaot=22.340000
```

Between the character % and the conversion character, there may be:

- ☐ A minus sign: Denoting left adjustment of the data.
- ☐ A digit: Specifying the minimum width in which the data is to be output, if the data has a larger number of characters then the specified width occupied by the output is larger. If the data consists of fewer characters then the specified width, it is padded to the right or to the left (if minus sign is not specified) with blanks. If the digit is prefixed with a zero, the padding is done with zeros instead of blanks.
- ☐ A period: Separating the width from the next digit.

- A digit following the period: specifying the precision (number of decimal places for numeric data) or the maximum number of characters to be output.
- Letter l: To indicate that the data item is a long integer and not an int.

Format specification string / Control string	Data	Output
%2d	9	9
%2d	123	123
%03d	9	009
%-2d	7	7
%5.3d	2	002
%3.1d	15	15
%3.5d	15	0015
%5s	“Output sting”	Output string
%15s	“Output sting”	Output string
%-15s	“Output sting”	Output string
%15.5s	“Output sting”	Output string
%.5s	“Output sting”	Output
%15.5s	“Output sting”	Output
%f	87.65	87.650000
%.4.1s	87.65	87.71

Example based on the conversion character:

```
#include<stdio.h>
#include<conio.h>
main()
{
    int num=65;

    printf(“Value of num is : %d\n”, num);

    printf(“Character equivalent of %d is %c\n”, num , num);
    getch();
    clrscr();
    rerurn o;
}
```

```
Value of num is : 65
Character equivalent of 65 is A
```

Formatted Input

The function `scanf()` is used for formatted input from standard input and provides many of the conversion facilities of the function `printf()`.

Syntax

```
scanf (format, num1, num2,.....);
```

The function `scanf()` reads and converts characters from the standard input depending on the format specification string and stores the input in memory locations represented by the other arguments (`num1, num2,....`).

For Example:

```
scanf(“ %c %d”,&Name, &Roll No);
```

Note: the data names are listed as `&Name` and `&Roll No` instead of `Name` and `Roll No` respectively. This is how data names are specified in a `scanf()` function. In case of string type data names, the data name is not preceded by the character `&`.

Example with program

Write a function to accept and display the element number and the weight of a proton. The element number is an integer and weight is fractional.

```
#include<stdio.h>
#include<conio.h>
main()
{
    int e_num;
    float e_wt;
    printf (“Enter the Element No. and Weight of a Proton\n”);
    scanf (“%d %f”,&e_num, &e_wt);

    printf (“The Element No.is:”,e_num);
    printf (“The Weight of a Proton is: %f\n”, e_wt);
    getch();
    return 0;
}
```

STORAGE CLASSES ☺

Every Variable in a program has memory associated with it.

Memory Requirement of Variables is different for different types of variables.

In C, Memory is allocated & released at different places

Term	Definition
Scope	Region or Part of Program in which Variable is accessible
Extent	Period of time during which memory is associated with variable
Storage Class	
	Manner in which memory is allocated by the Compiler for Variable

Storage class of variable Determines following things

Where the variable is stored

Scope of Variable

Default initial value

Lifetime of variable

A. Where the variable is stored:

Storage Class determines the location of variable, where it is declared. Variables declared with auto storage classes are declared inside main memory whereas variables declared with keyword register are stored inside the CPU Register.

B. Scope of Variable

Scope of Variable tells compile about the visibility of Variable in the block. Variable may have Block Scope, Local Scope and External Scope. A scope is the context within a computer program in which a variable name or other identifier is valid and can be used, or within which a declaration has effect.

C. Default Initial Value of the Variable

Whenever we declare a Variable in C, garbage value is assigned to the variable. Garbage Value may be considered as initial value of the variable. C Programming have different storage classes which has different initial values such as Global Variable have Initial Value as 0 while the Local auto variable have default initial garbage value.

D. Lifetime of variable

Lifetime of the = Time Of variable Declaration - Time of Variable Destruction

Suppose we have declared variable inside main function then variable will be destroyed only when the control comes out of the main .i.e end of the program.

Different Storage Classes:

Auto Storage Class

Static Storage Class

Extern Storage Class

Register Storage Class

Automatic (Auto) storage class

This is default storage class

All variables declared are of type Auto by default

In order to Explicit declaration of variable use 'auto' keyword

auto int num1 ; // Explicit Declaration

Features:

Storage	Memory
Scope	Local / Block Scope
Life time	Exists as long as Control remains in the block
Default initial Value	Garbage

Example

```
void main()
{
    auto num = 20 ;
    {
        auto num = 60 ;
        printf("nNum : %d",num);
    }
    printf("nNum : %d",num);
}
```

Num : 60

Num : 20

Note :

Two variables are declared in different blocks , so they are treated as different variables

External (extern) storage class in C Programming

Variables of this storage class are “Global variables”

Global Variables are declared outside the function and are accessible to all functions in the program

Generally, External variables are declared again in the function using keyword extern

In order to Explicit declaration of variable use ‘extern’ keyword

extern int num1 ; // Explicit Declaration

Features :

Storage	Memory
Scope	Global / File Scope
Life time	Exists as long as variable is running Retains value within the function
Default initial Value	Zero

Example

```
int num = 75 ;
void display();
void main()
{
    extern int num ;
    printf("nNum : %d",num);
    display();
}

void display()
{
    extern int num ;
```

```
        printf("\nNum : %d",num);  
    }
```

Output :

```
Num : 75
```

```
Num : 75
```

Note :

Declaration within the function indicates that the function uses external variable

Functions belonging to same source code , does not require declaration (no need to write extern)

If variable is defined outside the source code , then declaration using extern keyword is required.

Static Storage Class

The **static** storage class instructs the compiler to keep a local variable in existence during the life-time of the program instead of creating and destroying it each time it comes into and goes out of scope. Therefore, making local variables static allows them to maintain their values between function calls.

The static modifier may also be applied to global variables. When this is done, it causes that variable's scope to be restricted to the file in which it is declared.

In C programming, when **static** is used on a class data member, it causes only one copy of that member to be shared by all the objects of its class.

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

```
/* function declaration */
```

```
void func(void);
```

```
static int count = 5; /* global variable */
```

```
main() {
```

```
    while(count-->0) {
```

```
        func();
```

```
    }
```

```
    return 0;
```

```
}
```

```
/* function definition */
```

```
void func( void ) {
```

```
    static int i = 5; /* local static variable */
```

```
    i++;
```

```
printf("i is %d and count is %d\n", i, count);  
}
```

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

```
i is 6 and count is 4  
i is 7 and count is 3  
i is 8 and count is 2  
i is 9 and count is 1  
i is 10 and count is 0
```

Register Storage Class

register keyword is used to define local variable.

Local variable are stored in register instead of **RAM**.

As variable is stored in register, the **Maximum size of variable = Maximum Size of Register**
unary operator [&] is not associated with it because Value is not stored in RAM instead it is stored in Register.

This is generally used for **faster access**. Common use is “**Counter**”

Syntax

```
register int count;
```

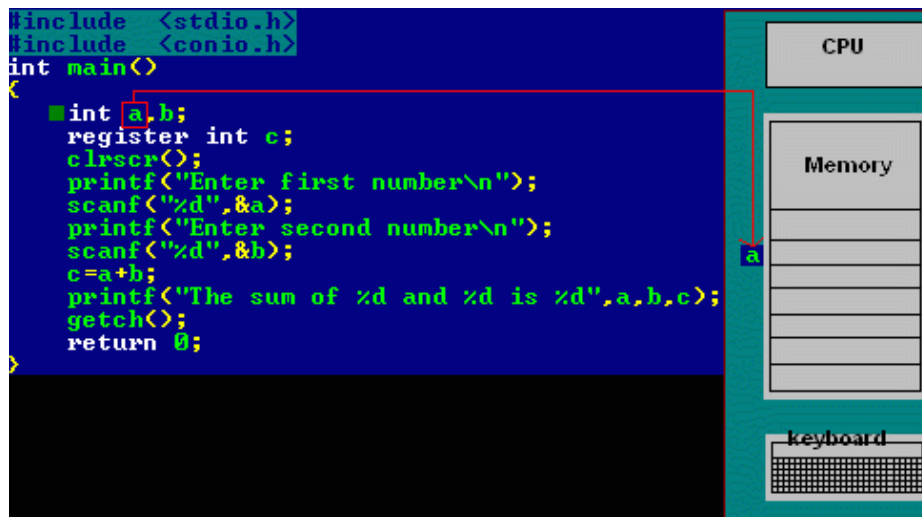
Register storage classes example

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

```
int main()  
{  
int a,b;  
register int c;  
clrscr();  
printf("\nEnter first number\n");  
scanf("%d",&a);  
printf("\nEnter second number\n");  
scanf("%d",&b);  
c = a + b;  
printf("The sum of %d and %d is %d ",a,b,c);  
getch();  
return 0;  
}
```

Explanation of program

Refer below animation which depicts the register storage classes –



In the above program we have declared two variables num1,num2. These two variables are stored in RAM.

Another variable is declared which is stored in register variable. Register variables are stored in the register of the microprocessor. Thus memory access will be faster than other variables.

If we try to declare more register variables then it can treat variables as Auto storage variables as memory of microprocessor is fixed and limited.

Why we need Register Variable ?

Whenever we declare any variable inside C Program then memory will be randomly allocated at particular memory location.

We have to keep track of that memory location. We need to access value at that memory location using ampersand operator/Address Operator i.e (&).

If we store same variable in the register memory then we can access that memory location directly without using the Address operator.

Register variable will be accessed faster than the normal variable thus increasing the operation and program execution. Generally we use register variable as Counter.

Note : It is not applicable for arrays, structures or pointers.

Summary of register Storage class

Keyword	register
Storage Location	CPU Register
Initial Value	Garbage
Life	Local to the block in which variable is declared.
Scope	Local to the block.

C - COMMAND LINE ARGUMENTS ☺

It is possible to pass some values from the command line or from other shell scripts to your C programs when they are executed. These values are called **command line arguments** and many times they are important for your program especially when you want to control your program from outside instead of hard coding those values inside the code.

The command line arguments are handled using main() function arguments

argc , argv[]

where,

argc – Number of arguments in the command line including program name

argv[] – This is carrying all the arguments

In real time application, it will happen to pass arguments to the main program itself. These arguments are passed to the main () function while executing binary file from command line.

For example, when we compile a program (test.c), we get executable file in the name “test”. Now, we run the executable “test” along with 4 arguments in command line like below.

./test this is a program

Where,

argc		5
argv[0]	=	“test”
argv[1]	=	“this”
argv[2]	=	“is”
argv[3]	=	“a”
argv[4]	=	“program”
argv[5]	=	NULL

EXAMPLE PROGRAM FOR ARGV () AND ARGV() FUNCTIONS IN C:

```
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *argv[]) // command line arguments
{
    if(argc!=5)
    {
        printf("Arguments passed through command line " \ "not equal to 5");
        return 1;
    }
}
```

```
}  
  
printf("\n Program name : %s \n", argv[0]);  
printf("1st  arg   :   %s   \n",  argv[1]);  
printf("2nd  arg   :   %s   \n",  argv[2]);  
printf("3rd  arg   :   %s   \n",  argv[3]);  
printf("4th  arg   :   %s   \n",  argv[4]);  
printf("5th arg : %s \n", argv[5]);  
  
return 0;  
}
```

Program name : test

1st arg : this

2nd arg : is

3rd arg : a

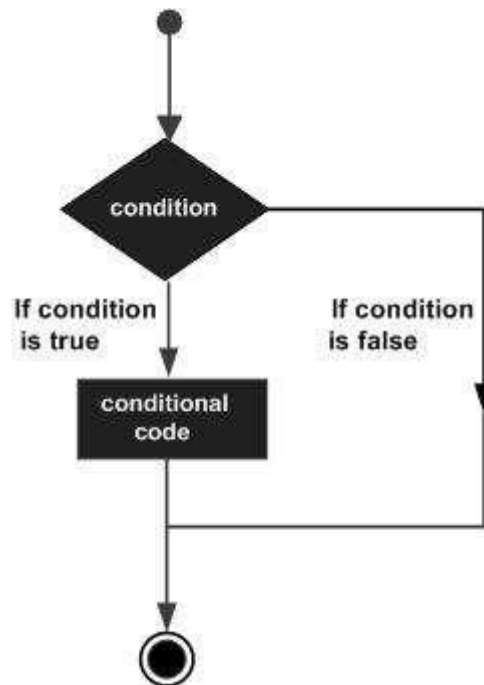
4th arg : program

5th arg : (null)

CONDITIONAL BRANCHING AND LOOPS | CONTROL STRUCTURES ☺

Decision-making structures require that the programmer specifies one or more conditions to be evaluated or tested by the program, along with a statement or statements to be executed if the condition is determined to be true, and optionally, other statements to be executed if the condition is determined to be false.

Shown below is the general form of a typical decision-making structure found in most of the programming languages:



C Programming language assumes any **non-zero** and **non-null** values as **true**, and if it is either **zero** or **null**, then it is assumed as **false** value.

C Programming language provides the following types of decision-making statements.

Statement	Description
if statement	An if statement consists of a Boolean expression followed by one or more statements.
if...else statement	An if statement can be followed by an optional else statement, which executes when the Boolean expression is false.

nested if statements	You can use one if or else if statement inside another if or else if statement(s).
switch statement	A switch statement allows a variable to be tested for equality against a list of values.
nested switch statements	You can use one switch statement inside another switch statement(s).

if Statement

An **if** statement consists of a Boolean expression followed by one or more statements.

Syntax

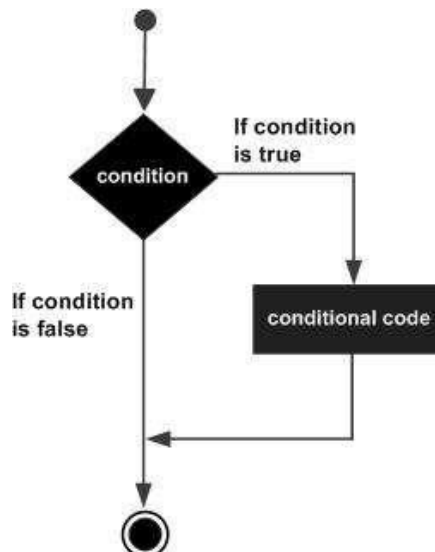
```
if(boolean_expression)
{
    /* statement(s) will execute if the boolean expression is true */
}
```

If the Boolean expression evaluates to **true**, then the block of code inside the 'if' statement will be executed. If the Boolean expression evaluates to **false**, then the first set of code after the end of the 'if' statement (after the closing curly brace) will be executed.

Note: For Single Line of Code – Opening and Closing braces are not needed.

C programming language assumes any **non-zero** and **non-null** values as **true** and if it is either **zero** or **null**, then it is assumed as **false** value.

Flow Diagram



Example

```
#include <stdio.h>

int main ()
{
    /* local variable definition */
    int a = 10;
    /* check the boolean condition using if statement */
    if( a < 20 )
    {
        /* if condition is true then print the following */
        printf("a is less than 20\n" );
    }
    printf("value of a is : %d\n", a);
    return 0;
}
```

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

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

if...else Statement

An **if** statement can be followed by an optional **else** statement, which executes when the Boolean expression is false.

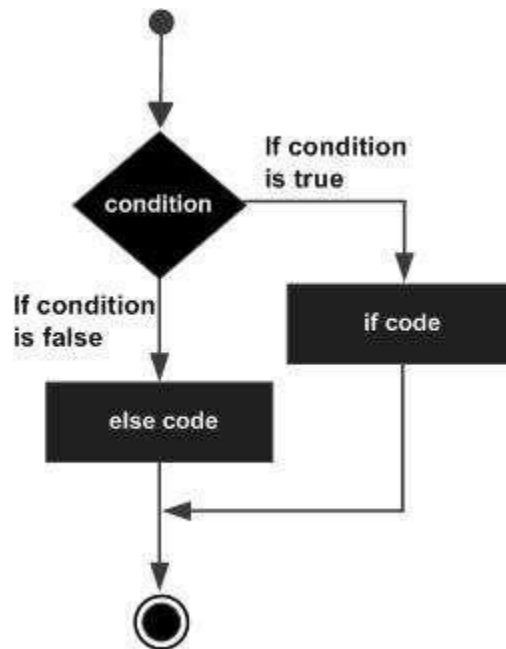
Syntax

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

If the Boolean expression evaluates to **true**, then the **if block** will be executed, otherwise, the **else block** will be executed.

C programming language assumes any **non-zero** and **non-null** values as **true**, and if it is either **zero** or **null**, then it is assumed as **false** value.

Flow Diagram



Example

```
#include <stdio.h>
int main ()
{
    /* local variable definition */
    int a = 100;
    /* check the boolean condition */
    if( a < 20 )
    {
        /* if condition is true then print the following */ printf("a is less than 20\n" );
    }
    else
    {
        /* if condition is false then print the following */ printf("a is not less than 20\n" );
    }
    printf("value of a is : %d\n", a);
    return 0;
}
```

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

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

if...else if...else Statement

An **if** statement can be followed by an optional **else if...else** statement, which is very useful to test various conditions using single if...else if statement.

When using if...else if...else statements, there are few points to keep in mind:

- ☐ An if can have zero or one else's and it must come after any else if's.
- ☐ An if can have zero to many else if's and they must come before the else.
- ☐ Once an else if succeeds, none of the remaining else if's or else's will be tested.

Syntax

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

```
if(boolean_expression 1)  
{  
    /* Executes when the boolean expression 1 is true */  
}  
else if( boolean_expression 2)  
{  
    /* Executes when the boolean expression 2 is true */  
}  
else if( boolean_expression 3)  
{  
    /* Executes when the boolean expression 3 is true */  
}  
else  
{  
    /* executes when the none of the above condition is true */  
}
```

Example

```
#include <stdio.h>

int main ()
{
    /* local variable definition */
    int a = 100;
    /* check the boolean condition */
    if( a == 10 )
    {
        /* if condition is true then print the following */
        printf("Value of a is 10\n" );
    }
    else if( a == 20 )
    {
        /* if else if condition is true */

        printf("Value of a is 20\n" );
    }
    else if( a == 30 )
    {
        /* if else if condition is true          */
        printf("Value of a is 30\n" );
    }
    else
    {
        /* if none of the conditions is true */ printf("None of the values is
        matching\n" );
    }
    printf("Exact value of a is: %d\n", a );
    return 0;
}
```

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

```
None of the values is matching
Exact value of a is: 100
```

Nested if Statements

It is always legal in C programming to **nest** if-else statements, which means you can use one if or else if statement inside another if or else if statement(s).

Syntax

The syntax for a **nested if** statement is as follows:

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

You can nest **else if...else** in the similar way as you have nested *if* statements.

Example

```
#include <stdio.h>
int main ()
{
    /* local variable definition */
    int a = 100;
    int b = 200;
    /* check the boolean condition */
    if( a == 100 )
    {
        /* if condition is true then check the following */
        if( b == 200 )
        {
            /* if condition is true then print the following */
            printf("Value of a is 100 and b is 200\n" );
        }
    }

    printf("Exact value of a is : %d\n", a );
    printf("Exact value of b is : %d\n", b );
    return 0;
}
```

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

Value of a is 100 and b is 200

Exact value of a is : 100

Exact value of b is : 200

Practice programs: ☺

1. Write a C program to find maximum between two numbers.
1. Write a C program to find maximum between three numbers.
2. Write a C program to check whether a number is negative, positive or zero.
3. Write a C program to check whether a number is divisible by 5 and 11 or not.
4. Write a C program to check whether a number is even or odd.
5. Write a C program to check whether a year is leap year or not.
6. Write a C program to check whether a character is alphabet or not.
7. Write a C program to input any alphabet and check whether it is vowel or consonant.
8. Write a C program to input any character and check whether it is alphabet, digit or special character.
9. Write a C program to check whether a character is uppercase or lowercase alphabet.
10. Write a C program to input week number and print week day.
11. Write a C program to input month number and print number of days in that month.
12. Write a C program to count total number of notes in given amount.
13. Write a C program to input angles of a triangle and check whether triangle is valid or not.
14. Write a C program to input all sides of a triangle and check whether triangle is valid or not.
15. Write a C program to check whether the triangle is equilateral, isosceles or scalene triangle.
16. Write a C program to find all roots of a quadratic equation.
17. Write a C program to calculate profit or loss.
18. Write a C program to input marks of five subjects Physics, Chemistry, Biology, Mathematics and Computer. Calculate percentage and grade according to following:
 - Percentage $\geq 90\%$: Grade A
 - Percentage $\geq 80\%$: Grade B
 - Percentage $\geq 70\%$: Grade C
 - Percentage $\geq 60\%$: Grade D
 - Percentage $\geq 40\%$: Grade E
 - Percentage $< 40\%$: Grade F

19. Write a C program to input basic salary of an employee and calculate its Gross salary according to following:

Basic Salary \leq 10000 : HRA = 20%, DA = 80%

Basic Salary \leq 20000 : HRA = 25%, DA = 90%

Basic Salary $>$ 20000 : HRA = 30%, DA = 95%

20. Write a C program to input electricity unit charges and calculate total electricity bill according to the given condition:

For first 50 units Rs. 0.50/unit

For next 100 units Rs. 0.75/unit

For next 100 units Rs. 1.20/unit

For unit above 250 Rs. 1.50/unit

An additional surcharge of 20% is added to the bill

switch Statement

A **switch** statement allows a variable to be tested for equality against a list of values. Each value is called a case, and the variable being switched on is checked for each **switch case**.

Why we should use Switch Case?

- ☐ One of the classic problem encountered in nested if-else / else-if ladder is called problem of Confusion.
- ☐ It occurs when no matching else is available for if .
- ☐ As the number of alternatives increases the Complexity of program increases drastically.
- ☐ To overcome this , C Provide a multi-way decision statement called 'Switch Statement'

Syntax

```
switch(expression){
```

```
    case constant-expression:
```

```
        statement(s);
```

```
        break; /* optional */
```

```
    case constant-expression:
```

```
        statement(s);
```



```
        break; /* optional */

/* you can have any number of case statements */

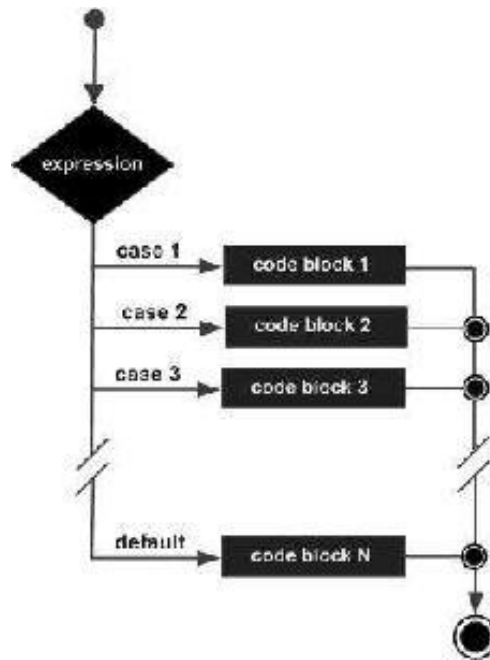
default : /* Optional */
    statement(s);

}
```

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

- ☐ The **expression** used in a **switch** statement must have an integral or enumerated type, or be of a class type in which the class has a single conversion function to an integral or enumerated type.
- ☐ You can have any number of case statements within a switch. Each case is followed by the value to be compared to and a colon.
- ☐ The **constant-expression** for a case must be the same data type as the variable in the switch, and it must be a constant or a literal.
- ☐ When the variable being switched on is equal to a case, the statements following that case will execute until a **break** statement is reached.
- ☐ When a **break** statement is reached, the switch terminates, and the flow of control jumps to the next line following the switch statement.
- ☐ Not every case needs to contain a **break**. If no **break** appears, the flow of control will *fall through* to subsequent cases until a break is reached.
- ☐ A **switch** statement can have an optional **default** case, which must appear at the end of the switch. The default case can be used for performing a task when none of the cases is true. No **break** is needed in the default case.

Flow Diagram



Example

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

```
int main ()
{
    /* local variable definition */
    char grade = 'B';

    switch(grade)
    {
        case 'A' :
            printf("Excellent!\n" );
            break;
        case 'B' :
            printf("Well done\n" );
            break;
        case 'C' :
            printf("Very good \n" );
            break;
        case 'D' :
            printf("You passed\n" );
            break;
        case 'F' :
            printf("Better try again\n" );
            break;
        default :
            printf("Invalid grade\n" );
    }
```

```

    }
    printf("Your grade is%c\n", grade );
    return 0;
}

```

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

```

Well done
Your grade is B

```

Nested switch Statements

It is possible to have a switch as a part of the statement sequence of an outer switch. Even if the case constants of the inner and outer switch contain common values, no conflicts will arise.

Syntax

The syntax for a **nested switch** statement is as follows:

```

switch(ch1) {
    case 'A':
        printf("This A is part of outer switch" );
        switch(ch2) {
            case 'A':
                printf("This A is part of inner switch" );
                break;
            case 'B':
                /* case code */
        }
        break;
    case 'B':
        /* case code */
}

```

Example

```

#include <stdio.h>
int main ()
{
    /* local variable definition */
    int a = 100;
    int b = 200;

```

```

switch(a) {
    case 100:
        printf("This is part of outer switch\n", a);
        switch(b) {
            case 200:
                printf("This is part of inner switch\n", a);
            }
        }
    printf("Exact value of a is : %d\n", a);
    printf("Exact value of b is : %d\n", b );
    return 0;
}

```

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

```

This is part of outer switch
This is part of inner switch
Exact value of a is : 100
Exact value of b is : 200

```

Practice Programs: 😊

1. Write a C program to print day of week name using switch case.
2. Write a C program print total number of days in a month using switch case.
3. Write a C program to check whether an alphabet is vowel or consonant using switch case.
4. Write a C program to find maximum between two numbers using switch case.
5. Write a C program to check whether a number is even or odd using switch case.
6. Write a C program to check whether a number is positive, negative or zero using switch case.
7. Write a C program to find roots of a quadratic equation using switch case.
8. Write a C program to create Simple Calculator using switch case.

The ? : Operator:

We have covered **conditional operator ? :** in the previous chapter which can be used to replace **if...else** statements. It has the following general form:

`Exp1 ? Exp2 : Exp3;`

Where Exp1, Exp2, and Exp3 are expressions. Notice the use and placement of the colon.
The value of a ? expression is determined like this:

1. Exp1 is evaluated. If it is true, then Exp2 is evaluated and becomes the value of the entire ? expression.
2. If Exp1 is false, then Exp3 is evaluated and its value becomes the value of the expression.

Example:

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

```
main() {  
    int a , b;  
    a = 10;  
    b=(a == 1) ? 20: 30;  
    printf( "Value of b is %d\n", (a == 1) ? 20: 30 );  
    printf( "Value of b is %d\n", (a == 10) ? 20: 30 );  
}
```

Value of b is 30
Value of b is 20

Pracrice programs: 😊

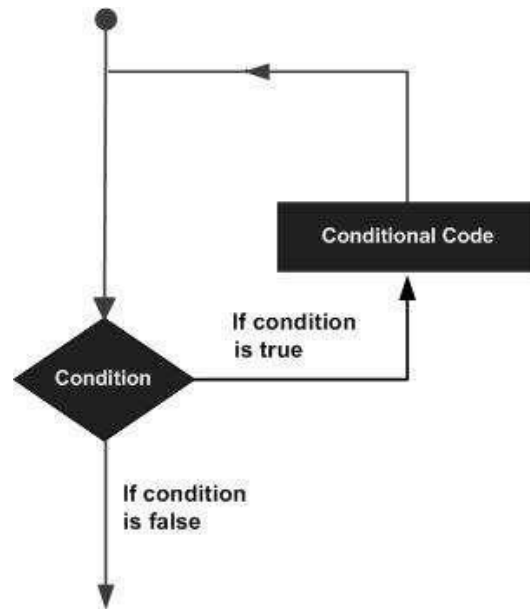
1. Write a C program to find maximum between two numbers using conditional operator.
2. Write a C program to find maximum between three numbers using conditional operator.
3. Write a C program to check whether a number is even or odd using conditional operator.
4. Write a C program to check whether year is leap year or not using conditional operator.
5. Write a C program to check whether character is an alphabet or not using conditional operator.

LOOP CONTROL STATEMENTS

We may encounter situations when a block of code needs to be executed several number of times. In general, statements are executed sequentially: The first statement in a function is executed first, followed by the second, and so on.

Programming languages provide various control structures that allow for more complicated execution paths.

A loop statement allows us to execute a statement or group of statements multiple times. Given below is the general form of a loop statement in most of the programming languages:



C programming language provides the following types of loops to handle looping requirements.

Loop Type	Description
while loop	Repeats a statement or group of statements while a given condition is true. It tests the condition before executing the loop body.
for loop	Executes a sequence of statements multiple times and abbreviates the code that manages the loop variable.
do...while loop	It is more like a while statement, except that it tests the condition at the end of the loop body.
nested loops	You can use one or more loops inside any other while, for, or do..while loop.

while Loop

A **while** loop in C programming repeatedly executes a target statement as long as a given condition is true. And it is a pretest loop.

Syntax

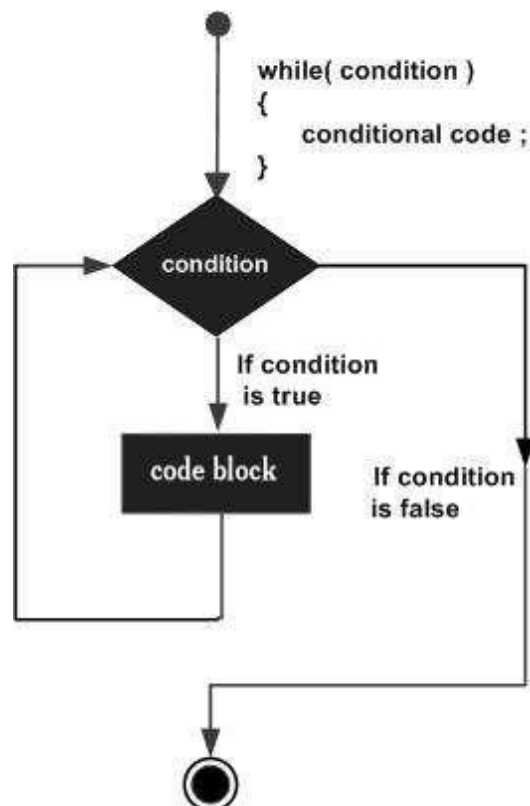
The syntax of a **while** loop in C programming language is:

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

Here, **statement(s)** may be a single statement or a block of statements. The **condition** may be any expression, and true is any nonzero value. The loop iterates while the condition is true.

When the condition becomes false, the program control passes to the line immediately following the loop.

Flow Diagram



Here, the key point to note is that a while loop might not execute at all, When the condition is tested and the result is false, the loop body will be skipped and the first statement after the while loop will be executed.

Example

```
#include <stdio.h>

int main ()
{
    /* local variable definition */
    int a = 10;
    /* while loop execution */
    while( a < 20 )
    {
        printf("value of a: %d\n", a);
        a++;
    }
    return 0;
}
```

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

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

for Loop

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

Syntax

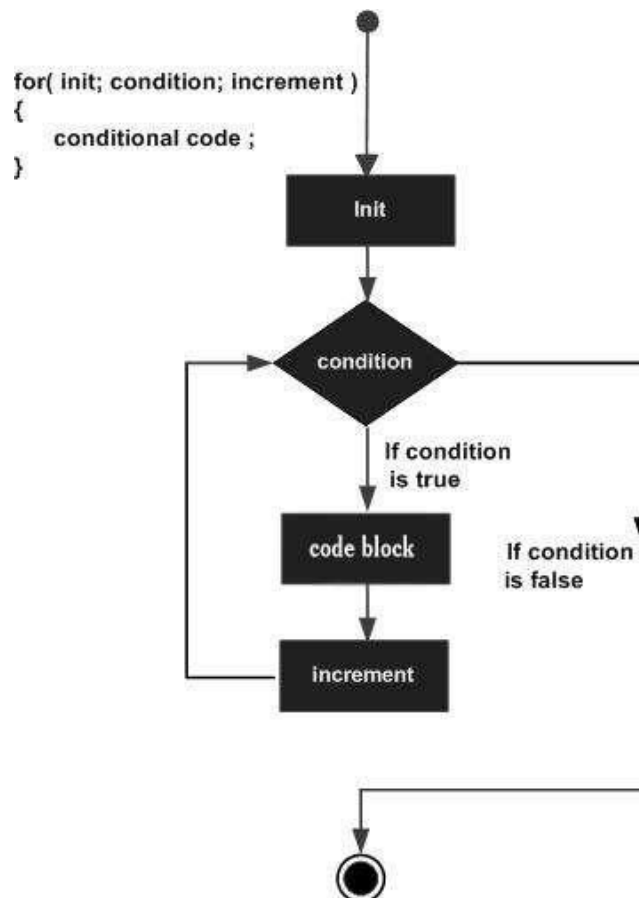
The syntax of a **for** loop in C programming language is:


```
for ( init; condition; increment/decrement)
{
    statement(s);
}
```

Here is the flow of control in a 'for' loop:

1. The **init** step is executed first, and only once. This step allows you to declare and initialize any loop control variables. You are not required to put a statement here, as long as a semicolon appears.
2. Next, the **condition** is evaluated. If it is true, the body of the loop is executed. If it is false, the body of the loop does not execute and the flow of control jumps to the next statement just after the 'for' loop.
3. After the body of the 'for' loop executes, the flow of control jumps back up to the **increment/decrement** statement. This statement allows you to update any loop control variables. This statement can be left blank, as long as a semicolon appears after the condition.
4. The condition is now evaluated again. If it is true, the loop executes and the process repeats itself (body of loop, then increment step, and then again condition). After the condition becomes false, the 'for' loop terminates.

Flow Diagram



Example

```
#include <stdio.h>

int main ()
{
    /* for loop execution */
    for( int a = 10; a < 20; a = a + 1 )
    {
        printf("value of a: %d\n", a);

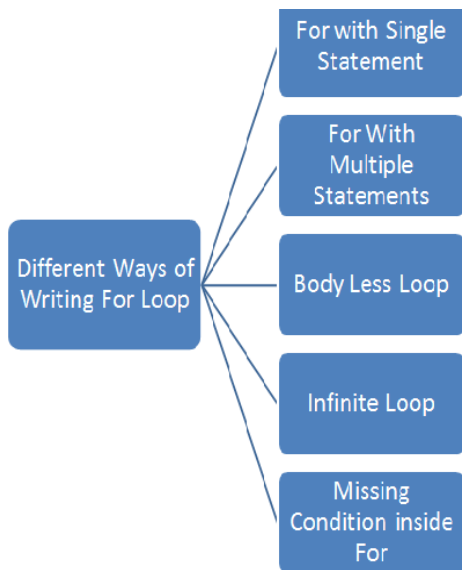
    }
    return 0;
}
```

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

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

For loop can be implemented in different verities of using for loop –

- Single Statement inside For Loop
- Multiple Statements inside For Loop
- No Statement inside For Loop
- Semicolon at the end of For Loop
- Multiple Initialization Statement inside For
- Missing Initialization in For Loop
- Missing Increment/Decrement Statement
- Infinite For Loop



Way 1 : Single Statement inside For Loop

```
for(i=0;i<5;i++)  
    printf("Hello");
```

Above code snippet will print Hello word 5 times.

We have single statement inside for loop body.

No need to wrap printf inside opening and closing curly block.
Curly Block is Optional.

Way 2 : Multiple Statements inside For Loop

```
for(i=0;i<5;i++)  
{  
    printf("Statement 1");  
    printf("Statement 2");  
    printf("Statement 3");
```

```
    if(condition)  
    {  
        -----  
        -----  
    }  
}
```

If we have block of code that is to be executed multiple times then we can use curly braces to wrap multiple statement in for loop.

Way 3 : No Statement inside For Loop

```
for(i=0;i<5;i++)  
{  
  
}
```

this is bodyless for loop. It is used to increment value of “i”. This variety of for loop is not used generally.

At the end of above for loop value of i will be 5.

Way 4 : Semicolon at the end of For Loop

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

Generally beginners thought that , we will get compile error if we write semicolon at the end of for loop.

This is perfectly legal statement in C Programming. This statement is similar to bodyless for loop. (Way 3)

Way 5 : Multiple Initialization Statement inside For

```
for(i=0,j=0;i<5;i++)  
{  
    statement1;  
    statement2;  
    statement3;  
}
```

Multiple initialization statements must be separated by Comma in for loop.

Way 6 : Missing Increment/Decrement Statement

```
for(i=0;i<5;)   
{  
    statement1;  
    statement2;  
    statement3;  
    i++;  
}
```

however we have to explicitly alter the value i in the loop body.

Way 7 : Missing Initialization in For Loop

```
i = 0;
for(;i<5;i++)
{
    statement1;
    statement2;
    statement3;
}
```

we have to set value of 'i' before entering in the loop otherwise it will take garbage value of 'i'.

Way 8 : Infinite For Loop

```
i = 0;
for(;;)
{
    statement1;
    statement2;
    statement3;
    if(breaking condition)
        break;
    i++;
}
```

Infinite for loop must have breaking condition in order to break for loop. Otherwise it will cause overflow of stack.

Summary of Different Ways of Implementing For Loop

Form	Comment
for (i=0 ; i < 10 ; i++) Statement1;	Single Statement
for (i=0 ; i <10; i++) { Statement1; Statement2; Statement3; }	Multiple Statements within for

for (i=0 ; i < 10;i++) ;	For Loop with no Body (Carefully Look at the Semicolon)
for (i=0,j=0;i<100;i++,j++) Statement1;	Multiple initialization & Multiple Update Statements Separated by Comma
for (; i<10 ; i++)	Initialization not used
for (; i<10 ;)	Initialization & Update not used
for (; ;)	Infinite Loop, Never Terminates

do...while Loop

Unlike **for** and **while** loops, which test the loop condition at the top of the loop, the **do...while** loop in C programming checks its condition at the bottom of the loop(post test 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

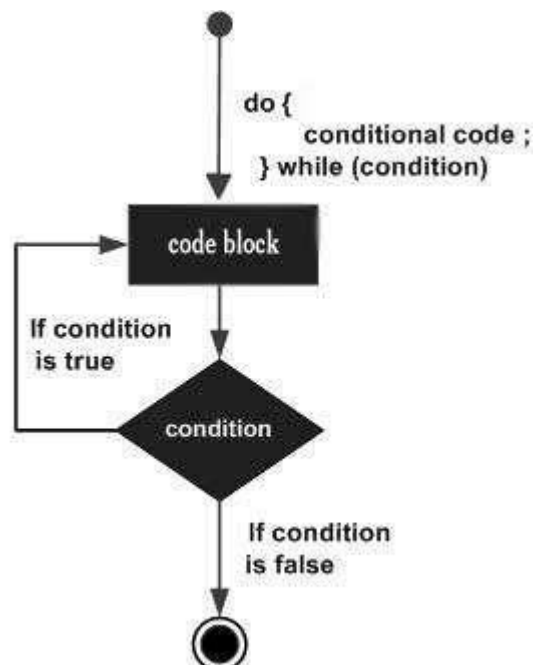
The syntax of a **do...while** loop in C programming language is:

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

Notice that the conditional expression appears at the end of the loop, so the statement(s) in the loop executes once before the condition is tested.

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

Flow Diagram



Example

```
#include <stdio.h>

int main ()
{
    /* local variable definition */
    int a = 10;

    /* do loop execution */
    do
    {
        printf("value of a: %d\n", a);
        a = a + 1;
    }while( a < 20 );
    return 0;
}
```

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

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

Nested Loops

C programming allows to use one loop inside another loop. The following section shows a few examples to illustrate the concept.

Syntax

The syntax for a **nested for loop** statement in C is as follows:

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

    statement(s);
}
```


The syntax for a **nested while loop** statement in C programming language is as follows:

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

    statement(s);
}
```

The syntax for a **nested do...while loop** statement in C programming language is as follows:

```
do
{
    statement(s);

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

}while( condition );
```

A final note on loop nesting is that you can put any type of loop inside any other type of loop. For example, a ‘for’ loop can be inside a ‘while’ loop or vice versa.

Example

The following program uses a nested for loop to find the prime numbers from 2 to 100:

```
#include <stdio.h>

int main ()
{
    /* local variable definition */
    int i, j;
    for(i=2; i<100; i++) {
        for(j=2; j <= (i/j); j++)
            if(!(i%j)) break;          // if factor found, not prime
        if(j > (i/j)) printf("%d is prime\n", i);
    }
    return 0;
}
```

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

```
2 is prime
3 is prime
5 is prime
7 is prime
11 is prime
13 is prime
17 is prime
19 is prime
23 is prime
29 is prime
31 is prime
37 is prime
41 is prime
43 is prime
47 is prime
53 is prime
59 is prime
61 is prime
67 is prime
71 is prime
73 is prime
79 is prime
83 is prime
89 is prime
97 is prime
```

Practice Programs: 😊

1. Write a C program to print all natural numbers from 1 to n. - using while loop
2. Write a C program to print all natural numbers in reverse (from n to 1). - using while loop
3. Write a C program to print all alphabets from a to z. - using while loop
4. Write a C program to print all even numbers between 1 to 100. - using while loop
5. Write a C program to print all odd number between 1 to 100.
6. Write a C program to find sum of all natural numbers between 1 to n.
7. Write a C program to find sum of all even numbers between 1 to n.
8. Write a C program to find sum of all odd numbers between 1 to n.
9. Write a C program to print multiplication table of any number.
10. Write a C program to count number of digits in a number.
11. Write a C program to find first and last digit of a number.
12. Write a C program to find sum of first and last digit of a number.
13. Write a C program to swap first and last digits of a number.
14. Write a C program to calculate sum of digits of a number.
15. Write a C program to calculate product of digits of a number.
16. Write a C program to enter a number and print its reverse.
17. Write a C program to check whether a number is palindrome or not.

18. Write a C program to find frequency of each digit in a given integer.
19. Write a C program to enter a number and print it in words.
20. Write a C program to print all ASCII character with their values.
21. Write a C program to find power of a number using for loop.
22. Write a C program to find all factors of a number.
23. Write a C program to calculate factorial of a number.
24. Write a C program to find HCF (GCD) of two numbers.
25. Write a C program to find LCM of two numbers.
26. Write a C program to check whether a number is Prime number or not.
27. Write a C program to print all Prime numbers between 1 to n.
28. Write a C program to find sum of all prime numbers between 1 to n.
29. Write a C program to find all prime factors of a number.
30. Write a C program to check whether a number is Armstrong number or not.
31. Write a C program to print all Armstrong numbers between 1 to n.
32. Write a C program to check whether a number is Perfect number or not.
33. Write a C program to print all Perfect numbers between 1 to n.
34. Write a C program to check whether a number is Strong number or not.
35. Write a C program to print all Strong numbers between 1 to n.
36. Write a C program to print Fibonacci series up to n terms.
37. Write a C program to find one's complement of a binary number.
38. Write a C program to find two's complement of a binary number.
39. Write a C program to convert Binary to Octal number system.
40. Write a C program to convert Binary to Decimal number system.
41. Write a C program to convert Binary to Hexadecimal number system.
42. Write a C program to convert Octal to Binary number system.
43. Write a C program to convert Octal to Decimal number system.
44. Write a C program to convert Octal to Hexadecimal number system.
45. Write a C program to convert Decimal to Binary number system.
46. Write a C program to convert Decimal to Octal number system.
47. Write a C program to convert Decimal to Hexadecimal number system.
48. Write a C program to convert Hexadecimal to Binary number system.
49. Write a C program to convert Hexadecimal to Octal number system.
50. Write a C program to convert Hexadecimal to Decimal number system.
51. Write a C program to print Pascal triangle upto n rows.
52. Star pattern programs - Write a C program to print the given star patterns.
53. Number pattern programs - Write a C program to print the given number patterns.

Loop Control Statements | Jump Statements

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

C supports the following control statements.

Control Statement	Description
break statement	Terminates the loop or switch statement and transfers execution to the statement immediately following the loop or switch.
continue statement	Causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating.
goto statement	Transfers control to the labeled statement.

break Statement

The **break** statement in C programming has the following two usages:

- When a **break** statement is encountered inside a loop, the loop is immediately terminated and the program control resumes at the next statement following the loop.
- It can be used to terminate a case in the **switch** statement (covered in the next chapter).

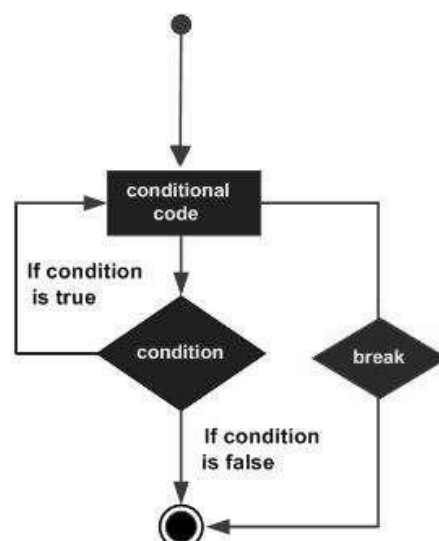
If you are using nested loops, the break statement will stop the execution of the innermost loop and start executing the next line of code after the block.

Syntax

The syntax for a **break** statement in C is as follows:

```
break;
```

Flow Diagram



Example

```
#include <stdio.h>

int main ()
{
    /* local variable definition */
    int a = 10;
    /* while loop execution */
    while( a < 20 )
    {
        printf("value of a: %d\n", a);
        a++;
        if( a > 15)
        {
            /* terminate the loop using break statement */
            break;
        }
    }
    return 0;
}
```

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

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

continue Statement

The **continue** statement in C programming works somewhat like the **break** statement. Instead of forcing termination, it forces the next iteration of the loop to take place, skipping any code in between.

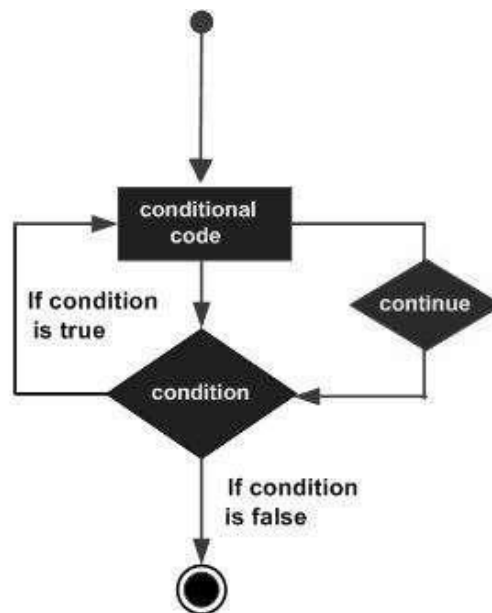
For the **for** loop, **continue** statement causes the conditional test and increment portions of the loop to execute. For the **while** and **do...while** loops, **continue** statement causes the program control to pass to the conditional tests.

Syntax

The syntax for a **continue** statement in C is as follows:

```
continue;
```

Flow Diagram



Example

```
#include <stdio.h>

int main ()
{
    /* local variable definition */
    int a = 10;
    /* do loop execution */
    do
    {
        if( a == 15)
        {
            /* skip the iteration */
            a = a + 1;
            continue;
        }
        printf("value of a: %d\n", a);
        a++;
    }while( a < 20 );
    return 0;
}
```

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

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

goto Statement

A **goto** statement in C programming provides an unconditional jump from the 'goto' to a labeled statement in the same function.

NOTE: Use of **goto** statement is highly discouraged in any programming language because it makes difficult to trace the control flow of a program, making the program hard to understand and hard to modify. Any program that uses a goto can be rewritten to avoid them.

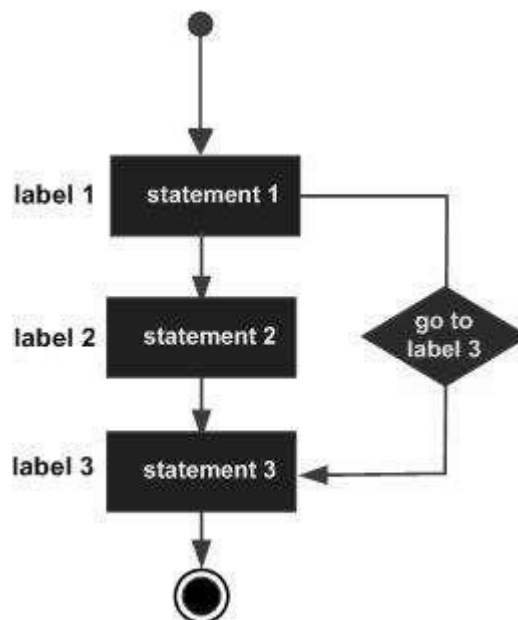
Syntax

The syntax for a **goto** statement in C is as follows:

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

Here **label** can be any plain text except C keyword and it can be set anywhere in the C program above or below to **goto** statement.

Flow Diagram



Example

```
#include <stdio.h>  
int main ()  
{  
    /* local variable definition */  
    int a = 10;  
  
    /* do loop execution */  
LOOP:  
do  
{  
    if( a == 15)  
    {
```

```

        /* skip the iteration */
        a = a + 1;
        goto LOOP;
    }
    printf("value of a: %d\n", a);
    a++;
}while( a < 20 );
return 0;
}

```

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

```

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

```

The Infinite Loop

A loop becomes an infinite loop if a condition never becomes false. The **for** loop is traditionally used for this purpose. Since none of the three expressions that form the ‘for’ loop are required, you can make an endless loop by leaving the conditional expression empty.

```

#include <stdio.h>

int main ()
{
    for( ; ; )
    {
        printf("This loop will run forever.\n");
    }
    return 0;
}

```

When the conditional expression is absent, it is assumed to be true. You may have an initialization and increment expression, but C programmers more commonly use the for(;;) construct to signify an infinite loop.

NOTE: You can terminate an infinite loop by pressing Ctrl + C keys.

SHORT ANSWERS☺

1. what is a computer ?
2. what are input and output devices?
3. list basic operations performed by computer
4. what is CPU? State its function. (or) what is processor? State its function
5. List subunits of CPU
6. Define control unit.
7. What are primary and secondary memories?
8. Difference between RAM and ROM.
9. What are volatile and non volatile memories?
10. Define cache memory.
11. What is an Operating system ? list out its goals and functions. [2019 May]
12. Define a software .
13. What is the difference between system software and application software?
14. Find and remove the error in the following c statement `a>b?g=a:g=b;` [2019 Aug]
15. Give a note on iteration statements in c language. [2019 Aug]
16. Name any two secondary storage devices and mention their characteristics [2019 May]
17. What is a flowchart? Explain with one example [2018 Dec]
18. What is a flowchart ? how it is different from algorithm ? [2019 June]
19. Write an algorithm to find the number is prime or not. [2018 Dec]
20. Why C is structured programming language ? Justify [2018 Dec]
21. Write a difference between break and exit() in c.
22. What is the difference between break and continue?
23. Define bit and byte .what it's use in computer programming.
24. What is the difference between assignment and equality operators?
25. What is meant by pre-test and post-test loops?
26. Write a program to find largest of two numbers using ?:
27. Swap two numbers without temp.
28. Write a program for Fibonacci.
29. What is the difference between while and do-while.
30. What is the difference between for and while.
31. What is Ternary operator?
32. What is a Pre-processor?
33. Give brief note on storage class. [2019 June]

LONG ANSWERS☺

ALGORITHMS, FLOWCHART, PSUEDOCODE

1. Explain about Algorithm? (or)
Define an Algorithm and State Properties of it.
2. List out the advantages and disadvantages of algorithm.
3. Define an algorithm .explain with an example
4. Write an algorithm to find HCF of a two positive integers. [2019 Aug]
5. Write an algorithm to find maximum number in a given set [2019 May]
6. Write an algorithm for finding the sum of first 'N' natural numbers. hint natural number = $N(N+1)/2$
7. Explain about a Flowchart?
8. Explain the Symbols in a Flowchart?
9. Write an algorithm to find the roots of quadratic equation considering all cases. [2019 June]

DATATYPES

1. . Explain data types in 'C'?

CONSTANTS

1. Describe the different types of constants in C with example? (or) What are the rules for creating C constants explain with example?

VARIABLES

1. Define a variable, list the rules for declaring variable. Give valid and invalid examples. [2019 Aug]
2. What is a variable? Write the rules for constructing a variable.

OPERATORS

1. What is an operator and List different categories of C operators based on their functionality? Give examples?
2. Explain about different bitwise operators with examples. [2018 Dec]

TYPECONVERSIONS

1. Discuss the concept of type conversion [2019 May]
2. Explain the types of type conversions in C with example?
3. Explain implicit and explicit type conversions

FORMATTED I/O

1. Explain briefly about formatted I/O.

CONTROL STATEMENTS

1. Describe the various control statements in c. [2018 Dec]
2. Explain in detail about selection statements. (Or) Explain in detail about Decision making statements.(Or) Explain in detail about branching statements.
3. Explain briefly multi way selection statements with suitable examples.
4. What are looping statements? Explain looping statements with examples.

5. write down the significance of a break statement inside a switch statement [2019 May]
6. Distinguish between all loop statements along with flowchart and with an example program. [2019 June]

COMMANDLINE

1. What are command line arguments? Explain briefly. [2019 May]
2. What are command line arguments? Explain with a complete 'C' program. [2018 Dec]

STORAGECLASSES

1. List and explain storage classes. [2019 Aug]
2. List and explain various storage classes available in c and state the reason why register storage classes are frequently used [2019 May]

OTHER

1. Write a program to find whether the given no is armstrong or not. [2018 Dec]
2. Write a program in 'C' to check whether a given integer number is odd or even [2019 June]
3. Explain the terms stdin , stdout and stderr [2019 Aug]
4. What is a precedence and associativity in an expression? What is their need? [2019 May]
5. Define precedence and associativity? Give an example?

(Or)

Explain the hierarchy (priority) and associativity(clubbing)of operators in „C“ with example?

6. Discuss about expressions in detail.
7. Define an expression? How can you evaluate an expression?
8. Explain briefly about high-level and low-level languages.
9. What is an algorithm? Explain the steps involve in development of c programs.
10. Explain c tokens.
11. Explain about computer hardware and software

(Or)

List and explain the functions of various parts of computer hardware and software.

12. Write a program to accept integer and print digits using words .
13. Explain the history of C?
14. Explain list of computer languages. (or) Explain the programming languages?
15. What is the general structure of a 'C' program and explain with example?
16. Write a program to swap two numbers without using temp.
17. Write a program to generate Fibonacci numbers.
18. Explain Creation and Running of programs? (or) Describe how the Developers will write the programs? (Or) What are different steps followed in program development? (Or)

Explain typical steps for entering,compiling and executing 'C' programs. [2019 June]

Riyaz Mohammed

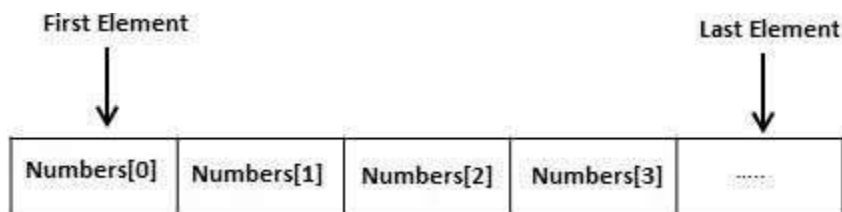
UNIT-II

ARRAYS

Arrays are a kind of data structure that can store a fixed-size sequential collection of elements of the same type. An array is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same type.

Instead of declaring individual variables, such as number0, number1, ..., and number99, you declare one array variable such as numbers and use numbers[0], numbers[1], and ..., numbers[99] to represent individual variables. A specific element in an array is accessed by an index.

All arrays consist of contiguous memory locations. The lowest address corresponds to the first element and the highest address to the last element.



Syntax :

```
<data type> <array name>[size of an array];
```

Subscript or indexing: A subscript is a property of an array that distinguishes all its stored elements because all the elements in an array have the same name (i.e. the array name). So to distinguish these, we use subscripting or indexing option.

E.g. int ar [20];

First element will be: int ar[0];

Second element will be: int ar[1];

Third element will be: int ar[2];

Fourth element will be: int ar[3];

Fifth element will be: int ar[4];

Sixth element will be: int ar[5];

So on.....

Last element will be: int ar[19];

- NOTE: An array always starts from 0 indexing.
- Example: `int ar[20];`

This above array will store 20 integer type values from 0 to 19.

ADVANTAGE OF AN ARRAY:

- Multiple elements are stored under a single unit.
- Searching is fast because all the elements are stored in a sequence.

APPLICATIONS OF USING ARRAY:

In c programming language, arrays are used in wide range of applications. Few of them are as follows...

• Arrays are used to Store List of values

In c programming language, single dimensional arrays are used to store list of values of same datatype. In other words, single dimensional arrays are used to store a row of values. In single dimensional array data is stored in linear form.

• Arrays are used to Perform Matrix Operations

We use two dimensional arrays to create matrix. We can perform various operations on matrices using two dimensional arrays.

• Arrays are used to implement Search Algorithms

We use single dimensional arrays to implement search algorithms like ...

Linear Search

Binary Search

• Arrays are used to implement Sorting Algorithms

We use single dimensional arrays to implement sorting algorithms like ...

Insertion Sort

Bubble Sort

Selection Sort

Quick Sort

Merge Sort, etc.,

• Arrays are used to implement Datastructures

We use single dimensional arrays to implement datastructures like...

Stack Using Arrays

Queue Using Arrays

• Arrays are also used to implement CPU Scheduling Algorithms

TYPES OF ARRAY

1. Static Array
2. Dynamic Array.

Static Array

An array with fixed size is said to be a static array.

Types of static array:

1. One Dimensional Array
2. Two Dimensional Arrays.
3. Multi Dimensional Array.

1. One Dimensional Array

An Array of elements is called 1 dimensional, which stores data in column or row form.

(Or) A list of items can be given one variable index is called single subscripted variable or a one-dimensional array

Declaration of an array:

We know that all the variables are declared before they are used in the program. Similarly, an array must be declared before it is used. During declaration, the size of the array has to be specified. The size used during declaration of the array informs the compiler to allocate and reserve the specified memory locations.

Syntax:

```
data_type array_name[size];
```

Where, size is the number of data items (or) index (or) dimension.
0 to (n-1) is the range of array.

Ex: int a[5];
 float x[10];

Initialization of Arrays:

The different types of initializing arrays:

1. At Compile time
 - a) Initializing all specified memory locations.
 - b) Partial array initialization
 - c) Initialization without size.
 - d) String initialization.

2. At Run Time

1. Compile Time Initialization

We can initialize the elements of arrays in the same way as the ordinary variables when they are declared. The general form of initialization of arrays is

datatype array-name[size]={ list of values};

- (a) Initializing all specified memory locations:-** Arrays can be initialized at the time of declaration when their initial values are known in advance. Array elements can be initialized with data items of type int, char etc.

Ex: - `int a[5]={ 10,15,1,3,20};`

During compilation, 5 contiguous memory locations are reserved by the compiler for the variable **a** and all these locations are initialized as shown in figure.

a[0]	a[1]	a[2]	a[3]	a[4]
10	15	1	3	20
1000	1002	1004	1006	1008

Fig: Initialization of int Arrays

Ex:-

`int a[3]={9,2,4,5,6}; //error: no. of initial values are more than the size of array.`

- (b) Partial array initialization:-** Partial array initialization is possible in c language. If the number of values to be initialized is less than the size of the array, then the elements will be initialized to zero automatically.

Ex:-

`int a[5]={ 10,15};`

Even though compiler allocates 5 memory locations, using this declaration statement; the compiler initializes first two locations with 10 and 15, the next set of memory locations are automatically initialized to 0's by compiler as shown in figure.

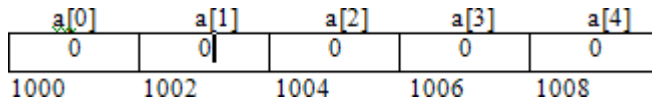
a[0]	a[1]	a[2]	a[3]	a[4]
10	15	0	0	0
1000	1002	1004	1006	1008

Fig: Partial Array Initialization

(c) Initialization with all zeros:-

Ex:-

```
int a[5]={0};
```



(d) Initialization without size:- Consider the declaration along with the initialization.

Ex:-

```
char b[]={'C','O','M','P','U','T','E','R'};
```

In this declaration, even though we have not specified exact number of elements to be used in array b, the array size will be set of the total number of initial values specified. So, the array size will be set to 8 automatically. The array b is initialized as shown in figure.

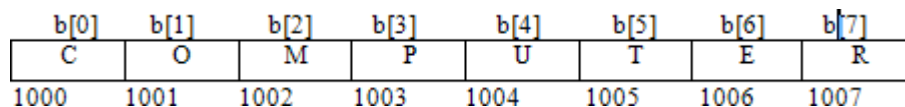


Fig: Initialization without size Ex:- `int ch[]={1,0,3,5} // array size is 4`

(e) Array initialization with a string:- Consider the declaration with string initialization. Ex:-

```
char b[]="COMPUTER";
```

The array b is initialized as shown in figure.

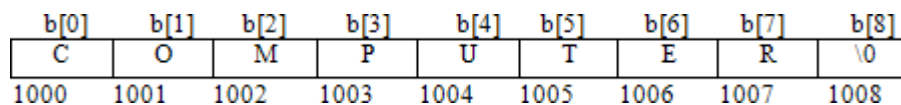


Fig: Array Initialized with a String

Even though the string "COMPUTER" contains 8 characters, because it is a string, it always ends with null character. So, the array size is 9 bytes (i.e., string length 1 byte for null character).

Ex:-

```
char b[9]={'C','O','M','P','U','T','E','R','\0'};// correct
```

```
char b[]={ 'C','O','M','P','U','T','E','R','\0'};// correct
```

```
char b[9]="COMPUTER";// correct
```

```
char b[8]="COMPUTER";// wrong
```


2. Run Time Initialization

An array can be explicitly initialized at run time. This approach is usually applied for initializing large arrays.

Ex: scanf can be used to initialize an array.

```
int x[3];
scanf("%d%d%d",&x[0],&x[1],&x[2]);
```

The above statements will initialize array elements with the values entered through the key board.

(Or)

By using loops .

```
for(i=0;i<100;i=i+1)
```

```
{
if(i<50)
sum[i]=0.0;
else
sum[i]=1.0;
}
```

The first 50 elements of the array sum are initialized to 0 while the remaining 50 are initialized to 1.0 at run time.

Assigning Values:

We can assign values to individual elements by using assignment operator (=)

eg:

```
int scores[6];
scores[4]=23;
```

We cannot assign one array to another array even if they match fully in type and size, but can be assigned individually.

Ex:

```
int scores1[5]={ 10,15,1,3,20};

int scores2[5]={ 100,150,10,30,20};
scores=scores; // wrong
```

But can be assigned individually.

```
scores1[0]=scores2[0];

scores1[1]=scores2[1];

scores1[2]=scores2[2];

.

.
```

```
scores1[4]=scores2[4];
```

By suing loops:

```
for(i=0;i<5;i++)  
{  
scores[i]=i+10; //we can also use scanf for dynamic values - scanf("%d",&scores[i]);  
}
```

Accessing Array Elements:

An element is accessed by indexing the array name. This is done by placing the index of the element within square brackets after the name of the array.

For example –

```
double salary = balance[9];
```

The above statement will take the 10th element from the array and assign the value to salary variable.

By loops:

```
for (j = 0; j < 10; j++ ) {  
    printf("Balance = %d\n",balance[j] );  
}
```

The following example shows how to use all the three above mentioned concepts viz. declaration, assignment, and accessing arrays –

```
#include <stdio.h>  
  
int main () {  
    int n[ 10 ]; /* n is an array of 10 integers */  
    int i,j;  
  
    /* initialize elements of array n to 0 */  
    for ( i = 0; i < 10; i++ ) {  
        n[ i ] = i + 100; /* set element at location i to i + 100 */  
    }  
    /* output each array element's value */  
    for (j = 0; j < 10; j++ ) {  
        printf("Element[%d] = %d\n", j, n[j] );  
    }  
    return 0;  
}
```

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

Element[0] = 100
Element[1] = 101
Element[2] = 102
Element[3] = 103
Element[4] = 104
Element[5] = 105
Element[6] = 106
Element[7] = 107
Element[8] = 108
Element[9] = 109

2. Two Dimensional Array.

An array of an array is said to be 2 dimensional array , which stores data in column and row form

(or):An array consisting of two subscripts is known as two-dimensional array. These are often known as array of the array. In two dimensional arrays the array is divided into rows and columns. These are well suited to handle a table of data. In 2-D array we can declare an array as :

Declaration:-Syntax:

data_type array_name[row_size][column_size];

Example: int a[3][4];

Where first index value shows the number of the rows and second index value shows the number of the columns in the array.

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

NOTE: In above example of two dimensional array, we have 3 rows and 4 columns.

NOTE: In above example of two dimensional array, we have total of 12 elements.

Initializing two-dimensional arrays:

Like the one-dimensional arrays, two-dimensional arrays may be initialized by following their declaration with a list of initial values enclosed in braces.

Ex: `int a[2][3]={0,0,0,1,1,1};` initializes the elements of the first row to zero and the second row to one. The initialization is done row by row.

The above statement can also be written as
`int a[2][3] = {{ 0,0,0},{1,1,1}};`
by surrounding the elements of each row by braces.

We can also initialize a two-dimensional array in the form of a matrix as shown below
`int a[2][3]={ {0,0,0},
 {1,1,1}
 };`

When the array is completely initialized with all values, explicitly we need not specify the size of the first dimension.

Ex: `int a[][3]={ {0,2,3},
 {2,1,2}
 };`

If the values are missing in an initializer, they are automatically set to zero.

Ex: `int a[2][3]={ {1,1},
 {2}
 };`

Will initialize the first two elements of the first row to one, the first element of the second row to two and all other elements to zero.

Assigning values to two dimensional array:

We can assign values to individual elements by using assignment operator (=) like below

```
int a[2][3];
```

```
a[0][0]=10;
```

```
a[0][1]=20;
```

```
a[0][2]=30;
```

```
a[1][0]=40;
```

```
a[1][1]=50;
a[1][2]=60;
```

Note:we can also assign values to arrays dynamically by using scanf() .

By Using loops:

```
for(i=0;i<2;i++)
{
    for (j = 0; j < 10; j++ )
    {
        a[i][j]=10*j;    //we can use  scanf("%d",a[i][j]) for dynamic input
    }
}
```

Accessing values from two dimensional array :

An element is accessed by indexing the array name. This is done by placing the indexes of the element within square brackets after the name of the array.

For example –

```
int num = a[0][1];
```

The above statement will take the 2nd element from the array and assign the value to num variable.

By Using loops:

```
for (i=0;i<2;i++)
{
    for (j = 0; j < 10; j++ )
    {
        printf("\n a[%d][%d]=%d",i,j,a[i][j]);
    }
}
```

3. Multi Dimensional Array.

This array does not exist in c and c++.

Dynamic Array.

This type of array also does not exist in c and c++.

Multidimensional arrays are often known as array of the arrays. In multidimensional arrays the array is divided into rows and columns, mainly while considering multidimensional arrays we will be discussing mainly about two dimensional arrays and a bit about three dimensional arrays.

Syntax: data_type array_name[size1][size2][size3]-----[sizeN]; In 2-D array we can declare an array as :

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

where first index value shows the number of the rows and second index value shows the number of the columns in the array. To access the various elements in 2-D array we can use:

```
printf("%d", arr[2][3]);
```

/* output will be 6, as arr[2][3] means third element of the second row of the array */ In 3-D we can declare the array in the following manner :

```
int arr[3][3][3] = {1, 2, 3,  
                   4, 5, 6,  
                   7, 8, 9,  
  
                   10, 11, 12,  
                   13, 14, 15,  
                   16, 17, 18,  
  
                   19, 20, 21,  
                   22, 23, 24,  
                   25, 26, 27  
                   };
```

/* here we have divided array into grid for sake of convenience as in above declaration we have created 3 different grids, each have rows and columns */

If we want to access the element the in 3-D array we can do it as follows :

```
printf("%d",arr[2][2][2]);
```

/* its output will be 27, as arr[2][2][2] means first value in [] corresponds to the grid no. i.e. 3 and the second value in [] means third row in the corresponding grid and last [] means third column*/

Ex:-

```
int arr[3][5][12];
```

```
float table[5][4][5][3];
```

arr is 3D array declared to contain 180 ($3*5*12$) int type elements. Similarly table is a 4D array containing 300 elements of float type.

Example: Program based upon array:

WAP to store marks in 5 subjects for a student. Display marks in 2nd and 5th subject.

```
#include<stdio.h>
#include<conio.h>
void main()
{
int ar[5]; int i;
for(i=0;i<5;i++)
{
printf("\n Enter marks in %d subject:",i+1);
scanf("%d",&ar[i]);
}
printf("Marks in 2nd subject is: ",ar[1]);
printf("Marks in 5th subject is: ",ar[4]);

}
```

Write a C program that uses functions to perform the following:

i. Addition of Two Matrices(IMP)

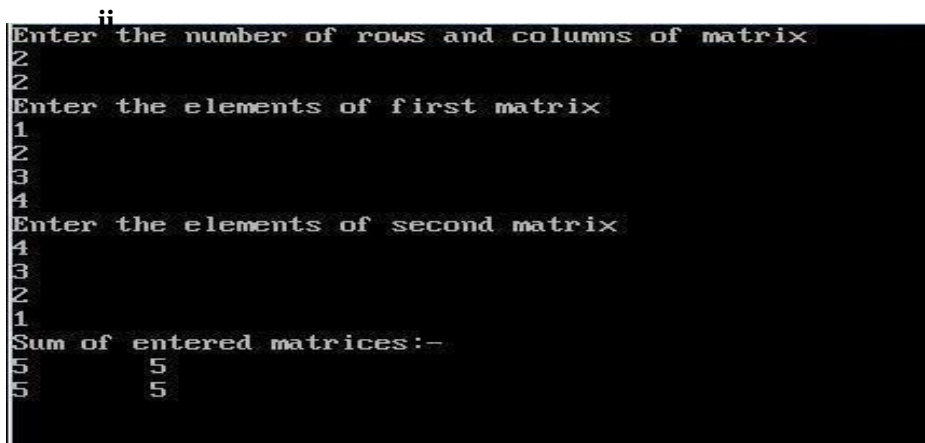
```
#include <stdio.h>
int main()
{
    int r, c, i, j, first[10][10], second[10][10], sum[10][10];
    clrscr();
    printf("Enter the number of rows and columns of matrix\n");
    scanf("%d%d", &r, &c);

    printf("Enter the elements of first matrix\n");
    for (i = 0; i < r; i++)
        for (j = 0; j < c; j++)
            scanf("%d", &first[i][j]);

    printf("Enter the elements of second matrix\n");
    for (i = 0; i < r; i++)
        for (j = 0; j < c; j++)
            scanf("%d", &second[i][j]);

    printf("Sum of entered matrices:-\n");
    for (i = 0; i < r; i++) {
        for (j = 0; j < c; j++) {
            sum[i][j] = first[i][j] + second[i][j];
            printf("%d\t", sum[i][j]);
        }
        printf("\n");
    }
    return 0;
}
```

Output:



The screenshot shows the execution of the C program. It prompts the user to enter the number of rows and columns of the matrix. The user enters 2 for rows and 2 for columns. Then, it prompts for the elements of the first matrix, and the user enters 1, 2, 3, 4. Next, it prompts for the elements of the second matrix, and the user enters 4, 3, 2, 1. Finally, it displays the sum of the two matrices, which is a 2x2 matrix with values 5, 5, 5, 5.

```
ii
Enter the number of rows and columns of matrix
2
2
Enter the elements of first matrix
1
2
3
4
Enter the elements of second matrix
4
3
2
1
Sum of entered matrices:-
5      5
5      5
```


ii. Multiplication of Two Matrices(IMP)

```
#include <stdio.h>

int main()
{
    int r1, r2, c1, c2, i, j, k, sum = 0;
    int first[10][10], second[10][10], multiply[10][10];

    printf("Enter number of rows and columns of first matrix\n");
    scanf("%d%d", &r1, &c1);

    printf("Enter elements of first matrix\n");
    for (i = 0; i < r1; i++)
        for (j = 0; j < c1; j++)
            scanf("%d", &first[i][j]);

    printf("Enter number of rows and columns of second matrix\n");
    scanf("%d%d", &r2, &c2);

    if (c1 != r2)
        printf("The matrices can't be multiplied with each other.\n");
    else
    {
        printf("Enter elements of second matrix\n");
        for (i = 0; i < r2; i++)
            for (j = 0; j < c2; j++)
                scanf("%d", &second[i][j]);

        for (i = 0; i < r1; i++)
        {
            for (j = 0; j < c2; j++)
            {
                for (k = 0; k < c1; k++)
                {
                    multiply[i][j] = multiply[i][j] + first[i][k] * second[k][j];
                }
            }
        }

        printf("Product of the matrices:\n");
        for (i = 0; i < r1; i++) {
            for (j = 0; j < c2; j++)
                printf("%d\t", multiply[i][j]);
            printf("\n");
        }
    }
}
```

```

    }
    getch();
    return 0;
}

```

Output:

```

Enter number of rows and columns of first matrix
2 2
Enter elements of first matrix
1 2 3 4
Enter number of rows and columns of second matrix
2 2
Enter elements of second matrix
4
3
2
1
Product of the matrices:
8      5
20     13

```

iii. Transpose of a matrix with memory dynamically allocated for the new matrix as row and column counts may not be same

Program:

```

#include <stdio.h>
int main()
{
    int a[10][10], transpose[10][10], r, c, i, j;

    printf("Enter rows and columns of matrix: ");
    scanf("%d %d", &r, &c);

    // storing elements of the matrix
    printf("\nEnter elements of matrix:\n");
    for(i=0; i<r; ++i)
        for(j=0; j<c; ++j)
        {
            printf("Enter element a%d%d: ", i+1, j+1);
            scanf("%d", &a[i][j]);
        }

    // Displaying the matrix a[][] */
    printf("\nEnter Matrix: \n");
    for(i=0; i<r; ++i)

```

```

{
    for(j=0; j<c; ++j)
    {
        printf("%d ", a[i][j]);
    }
    printf("\n\n");
}
// Finding the transpose of
matrix a for(i=0; i<r; ++i)
    for(j=0; j<c; ++j)
    {
        transpose[j][i] = a[i][j];
    }
// Displaying the transpose of matrix a
printf("\nTranspose of Matrix:\n");
for(i=0; i<c; ++i)
{
    for(j=0; j<r; ++j)
    {
        printf("%d ", transpose[i][j]);
    }
    printf("\n\n");
}
return 0;
}

```

Output:

```

2
3
Enter elements of matrix:
Enter element a11: 1
Enter element a12: 2
Enter element a13: 3
Enter element a21: 4
Enter element a22: 5
Enter element a23: 6

Entered Matrix:
1 2 3
4 5 6

Transpose of Matrix:
1 4
2 5
3 6

```

C STRINGS:

In C language a string is group of characters (or) array of characters, which is terminated by delimiter \0 (null). Thus, C uses variable-length delimited strings in programs.

DECLARING STRINGS:

C does not support string as a data type. It allows us to represent strings as character arrays. In C, a string variable is any valid C variable name and is always declared as an array of characters.

Syntax:-

```
char string_name[size];
```

The size determines the number of characters in the string name.

Ex:- char city[10];
 char name[30];

INITIALIZING STRINGS:-

There are several methods to initialize values for string variables.

Ex:- char str[6]="HELLO";

H	E	L	L	O	\0
---	---	---	---	---	----

Ex:- char month[]="JANUARY";

J	A	N	U	A	R	Y	\0
---	---	---	---	---	---	---	----

Ex:- char city[8]="NEWYORK";

char city[8]={',N','E','W','Y','O','R','K','\0'};

The string city size is 8 but it contains 7 characters and one character space is for NULL terminator.

Storing strings in memory:-

In C a string is stored in an array of characters and terminated by \0 (null).

Ex:-

H	E	L	L	O	\0
---	---	---	---	---	----

 → delimiter

A string is stored in array, the name of the string is a pointer to the beginning of the string. The character requires only one memory location.

If we use one-character string it requires two locations. The difference is shown below,

H

a character

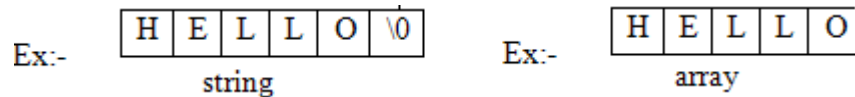
H	\0
---	----

one-character string

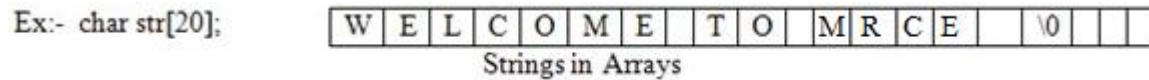
\0

empty string

The difference between array and string is shown below,



Because strings are variable-length structure, we must provide enough room for maximum-length string to store and one byte for delimiter.



Why do we need null?

A string is not a datatype but a data structure. String implementation is logical not physical. The physical structure is array in which the string is stored. The string is variable-length, so we need to identify logical end of data in that physical structure.

String constant (or) Literal:-

String constant is a sequence of characters enclosed in double quotes. When string constants are used in C program, it automatically initializes a null at end of string.

Ex:- "Hello" "Welcome" "Welcome to C Lab"

READING AND WRITING STRINGS:

C language provides several string handling functions for input and output.

STRING INPUT/OUTPUT FUNCTIONS:-

C provides two basic methods to read and write strings. Using formatted input/output functions and using a special set of functions.

Reading strings from terminal:-

- a) **formatted input function:-** scanf can be used with %s format specification to read a string.

Ex:- `char name[10];`

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

Here don't use „&“ because name of string is a pointer to array. The problem with scanf is that it terminates its input on the first white space it finds.

Ex:- NEW YORK

Reads only NEW (from above example).

b) **Unformatted input functions:-**

- (1) **getchar()**:- It is used to read a single character from keyboard. Using this function repeatedly we may read entire line of text

Ex:- char ch="z";

ch=getchar();

- (2) **gets()**:- It is more convenient method of reading a string of text including blank spaces. Ex:- char line[100];

gets(line);

Writing strings on to the screen:-

- a) **Using formatted output functions:-** printf with %s format specifier we can print strings in different formats on to screen.

Ex:- char name[10];

printf("%s",name);

Ex:- char name[10];

printf("%0.4",name);

J	A	N	U
---	---	---	---

/* If name is JANUARY prints only 4 characters ie. JANU */

Printf("%10.4s",name);

						J	A	N	U
--	--	--	--	--	--	---	---	---	---

printf("%-10.4s",name);

J	A	N	U						
---	---	---	---	--	--	--	--	--	--

b) **Using unformatted output functions:-**

- (1) **putchar()**:- It is used to print a character on the screen.

Ex:- putchar(ch);

- (2) **puts()**:- It is used to print strings including blank spaces.

Ex:- char line[15]="Welcome to lab";

puts(line);

STRING HANDLING FUNCTIONS ☺ (IMP)

C supports a number of string handling functions. All of these built-in functions are aimed at performing various operations on strings and they are defined in the header file **string.h**.

(i) strlen (ii) strcpy (iii) strcmp (iv) strcat (v)strupr() (vi)strlwr() (vii)strrev()

(i). strlen()

This function is used to find the length of the string excluding the NULL character. In other words, this function is used to count the number of characters in a string. Its syntax is as follows:

Int strlen(string);

Example: `char str1[] = "WELCOME";`
 `int n;`
 `n = strlen(str1);`

/* A program to calculate length of string by using strlen() function*/

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

char string1[50];
int length;
printf("\n Enter any string:");
gets(string1);
length=strlen(string1);
printf("\n The length of string=%d",length);
}
```

Output:

Enter any string: Dennis
The length of string=5

(ii). strcpy()

This function is used to copy one string to the other. Its syntax is as follows:

strcpy(string1,string2);

where string1 and string2 are one-dimensional character arrays.

This function copies the content of string2 to string1.

E.g., string1 contains master and string2 contains madam, then string1 holds madam after execution of the strcpy (string1,string2) function.

Example: char str1[] = "WELCOME";

```
char str2[ ] ="HELLO";  
strcpy(str1,str2);
```

/* A program to copy one string to another using strcpy() function */

```
#include<stdio.h>  
#include<conio.h>  
void main()  
{  
char  string1[30],string2[30];  
printf("\n Enter first string:");  
gets(string1);  
printf("\n Enter second string:");  
gets(string2);  
strcpy(string1,string2);  
printf("\nFirststring=%s",string1);  
printf("\nSecondstring=%s",string2);  
}
```

Output:

```
Enter first string: Computer  
Enter second string: Language  
Firststring= Language  
Secondstring= Language
```

(iii). strcmp ()

This function compares two strings character by character (ASCII comparison) and returns one of three values {-1,0,1}. The numeric difference is „0“ if strings are equal .If it is negative string1 is alphabetically above string2 .If it is positive string2 is alphabetically above string1.

Its syntax is as follows:

```
Int strcmp(string1,string2);
```


Example: char str1[] = "ROM";
 char str2[] = "RAM";
 strcmp(str1,str2);
 (or)
 strcmp("ROM","RAM");

/* A program to compare two strings using strcmp() function */

```
#include<stdio.h>
#include<conio.h>
void main()
{
char string1[30],string2[15];
int x;
printf("\n Enter first string:");
gets(string1);
printf("\n Enter second string:");
gets(string2);
x=strcmp(string1,string2);
if(x==0)
printf("\n Both strings are equal");
else if(x>0)
printf("\n First string is bigger");
else
printf("\n Second string is bigger");
}
```

Output:

```
Enter first string: dennis
Enter second string: Ritchie
Second string is bigger
```

(iv). strcat ()

This function is used to concatenate two strings. i.e., it appends one string at the end of the specified string. Its syntax as follows:

strcat(string1,string2);

where string1 and string2 are one-dimensional character arrays.

This function joins two strings together. In other words, it adds the string2 to string1 and

the string1 contains the final concatenated string. E.g., string1 contains **prog** and string2 contains **ram**, then string1 holds **program** after execution of the strcat() function.

Example: char str1[10] = “VERY”;

 char str2[5] =”GOOD”;

 strcat(str1,str2);

/* A program to concatenate one string with another using strcat() function*/

```
#include<stdio.h>
#include<string.h> main()
{
char string1[30],string2[15];
printf(“\n Enter first string:”);
gets(string1);
printf(“\n Enter second string:”);
gets(string2);
strcat(string1,string2);
printf(“\n Concatenated string=%s”,string1);
}
```

Output:

```
Enter first string: computer
Enter second string: lanuage
Concatenated string= computerlanuage
```

(v)strupr()

This string function is basically used for the purpose of converting the case sensitiveness of the string i.e. it converts string case sensitiveness into uppercase.

strupr(string);

Example: char str = “RiyazMohammed”

 strupr(str);

 printf(“The uppercase of the string is : %s”,str);

/* A program to convert a string to uppercase usingstrupr() function*/

```
#include<stdio.h>
#include<string.h> main()
{
char string1[30];
printf("\n Enter first string:");
gets(string1);
strupr(string1);
printf("\n Uppercase string=%s",string1);}
}
```

Output:

Enter first string: RiyazMohammed Uppercase string= RIYAZMOHAMMED
--

(vi) strlwr ()

This string function is basically used for the purpose of converting the case sensitiveness of the string i.e it converts string case sensitiveness into lowercase.

strlwr(string);

Example: char str = "RIYAZMOHAMMED"

```
strlwr(str);
printf("The Lowercase of the string is :%s ",str);
```

/* A program to convert a string to lowercase using strlwr() function*/

```
#include<stdio.h>
#include<string.h> main()
{
char string1[30];
printf("\n Enter first string:");
gets(string1);
strlwr(string1);
printf("\n Lowercase string=%s",string1);
}
```

Output:

Enter first string: RIYAZMOHAMMED
Lowercase string= riyazmohammed

(vii) strrev()

This string function is basically used for the purpose of reversing the string.

strrev(string);

```
char str1= "won";  
char str2[20];
```

```
str2= strrev(str1);  
printf("%s",str2);
```

/* A program to print reverse of a string using strrev() function*/

```
#include<stdio.h>  
#include<string.h> main()  
{  
  
char string1[30];  
printf("\n Enter first string:");  
gets(string1);  
strrev(string1);  
printf("\n Reversed string=%s",string1);  
}
```

Output:

Enter first string: won
Reversed string= now

WAP to accept a string and perform various operations:

1. To convert string into upper case.
2. To reverse the string .
3. To copy string into another string.
4. To compute length depending upon user choice.

```

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

void main()
{
char str[20]; char str1[20]; int opt,len;
printf("\n MAIN MENU");

printf("\n1. Convert string into upper case");
printf("\n2. Reverse the string");
printf("\n3. Copy one string into another string");
printf("\n4.Compute length of string ");
printf("Enter string: ");
scanf("%s",&str);
printf("Enter your choice:");
scanf("%d",&opt); switch(opt)
{
case 1:               strupr(str);
                        printf("The string in uppercase is :%s ",str);
                        break;

case 2:                strrev(str);
                        printf("The reverse of string is : %s",str);
                        break;

case 3:                strcpy(str1,str);
                        printf("New copied string is : %s",str1);
                        break;

case 4:                len=strlen(str);
                        printf("The length of the string is : %s",len);
                        break;

default:                printf("Ypu have entered a wrong choice.");
}
}

```

Output:

MAIN MENU

1. Convert string into upper case
2. Reverse the string
3. Copy one string into another string
- 4.Compute length of string

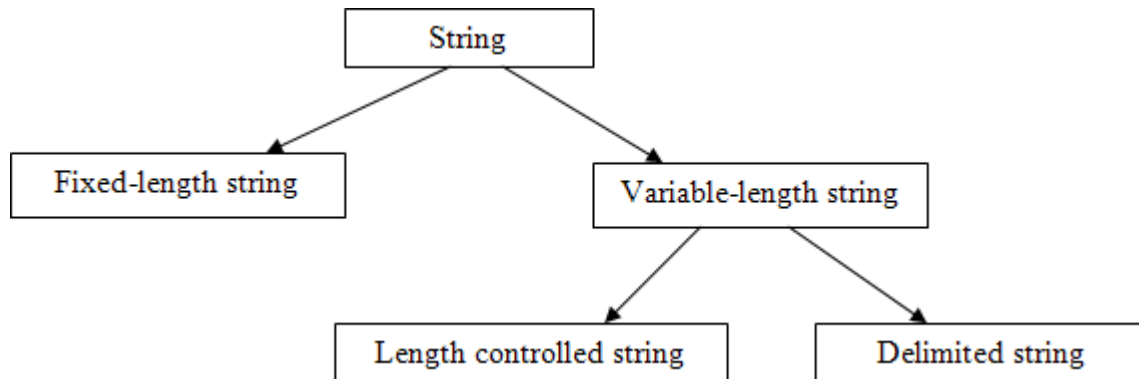
Enter string: now

Enter your choice: 2

The reverse of string is : won

FIXED AND VARIABLE LENGTH FORMAT STRINGS

In general a string is a series of characters (or) a group of characters. While implementation of strings, a string created in pascal differs from a string created in C language.



1. Fixed-length strings:

When implementing fixed-length strings, the size of the variable is fixed. If we make it too small we can't store, if we make it too big, then waste of memory. And another problem is we can't differentiate data (characters) from non-data (spaces, null etc).

2. Variable-length string:

The solution is creating strings in variable size; so that it can expand and contract to accommodate data. Two common techniques used,

(a) Length controlled strings:

These strings added a count which specifies the number of characters in the string.

Ex:-

5	H	E	L	L	O
---	---	---	---	---	---

(b) Delimited strings:

Another technique is using a delimiter at the end of strings. The most common delimiter is the ASCII null character (\0).

Ex:-

H	E	L	L	O	\0
---	---	---	---	---	----

ARRAY OF STRINGS :

We have array of integers, array of floating point numbers, etc.. Similarly we have array of strings also.

Collection of strings is represented using array of strings.

Declaration:-

Char arr[row][col];

where,

arr - name of the array
row - represents number of strings
col - represents size of each string

Initialization:-

char arr[row][col] = { list of strings };

Example:-

char city[5][10] = {"DELHI", "CHENNAI", "BANGALORE", "HYDERABAD", "MUMBAI"};

D	E	L	H	I	\0				
C	H	E	N	N	A	I	\0		
B	A	N	G	A	L	O	R	E	\0
H	Y	D	E	R	A	B	A	D	\0
M	U	M	B	A	I	\0			

In the above storage representation memory is wasted due to the fixed length for all strings.

```
#include<stdio.h>
#include<conio.h>
void main()
{
    char city[5][10]= {"DELHI", "CHENNAI", "BANGALORE", "HYDERABAD",
    "MUMBAI"};
    int i;
    clrscr();
    for (i=0;i<5;i++)
        printf("\n%s"city[i]);
}
```

```
DELHI
CHENNAI
BANGALORE
HYDERABAD
```

STRUCTURES

Definition: A structure is a collection of one or more variables of different data types, grouped together under a single name. By using structures variables, arrays, pointers etc can be grouped together.

Suppose Student record is to be stored, then for storing the record we have to group together all the information such as Roll, Name, and Percent which may be of different data types.

Ideally Structure is collection of different variables under single name. Basically Structure is for storing the complicated data.

A structure is a convenient way of grouping several pieces of related information

Structures can be declared using two methods as follows:

(i) Tagged Structure:

The structure definition associated with the structure name is referred as tagged structure. It doesn't create an instance of a structure and does not allocate any memory.

The **general form or syntax of tagged structure** definition is as follows,

struct TAG	Ex:-	struct student
{		{
Type variable1;		int htno[10];
Type variable2;		char name[20];
.....		float marks[6];
.....		};
Type variable-n;		
};		

Where,

- struct is the keyword which tells the compiler that a structure is being defined.
- Tag_name is the name of the structure.
- variable1, variable2 ... are called members of the structure.
- The members are declared within curly braces.
- The closing brace must end with the semicolon.
-

(ii) Type-defined structures:-

- The structure definition associated with the keyword **typedef** is called type-defined structure.
- This is the most powerful way of defining the structure.

The **syntax of typedefed structure** is

typedef struct

{

Type variable1;

Type variable2;

.....

.....

Type variable-n;

}Type;

Ex:-

typedef struct

{

int htno[10];

char name[20];

float marks[6];

}student;

where

- typedef is keyword added to the beginning of the definition.
- struct is the keyword which tells the compiler that a structure is being defined.
- variable1, variable2...are called fields of the structure.
- The closing brace must end with type definition name which in turn ends with semicolon.

Variable declaration:

Memory is not reserved for the structure definition since no variables are associated with the structure definition. The members of the structure do not occupy any memory until they are associated with the structure variables.

After defining the structure, variables can be defined as follows:

For first method,

struct TAG v1,v2,v3....vn;

For second method, which is most

powerful is, Type v1,v2,v3,....vn;

Alternate way:

struct TAG

{

Type variable1;

Type variable2;

.....

.....

Type variable-n;

} v1, v2, v3;

Ex:

```
struct book
{ char name[30]; int pages;
float price;
}b1,b2,b3;
```

Declare the C structures for the following scenario:

- i. College contains the following fields: College code (2characters), College Name, year of establishment, number of courses.
- ii. Each course is associated with course name (String), duration, number of students. (A College can offer 1 to 50 such courses)

(i). Structure definition for college :-

```
struct college
{
char code[2];
char college_name[20];
int year;
int no_of_courses;
};
```

Variable declaration for structure college :-

```
void main( )
{
struct college col1,col2,col3;
....
}
```

(ii). Structure definition for course :-

```
struct course
{
char course_name[20];
float duration;
int no_of_students;
};
```

Variable declaration for structure course :-

```
void main( )
{
struct course c1,c2,c3;
....
}
```

Initializing structures in C:

The rules for structure initialization are similar to the rules for array initialization. The initializers are enclosed in braces and separated by commas. They must match their corresponding types in the structure definition.

The syntax is shown below,

```
struct tag_name variable = {value1, value2,... value-n};
```

Structure initialization can be done in any one of the following ways

Initialization along with Structure definition:-

Consider the structure definition for student with three fields name, roll number and average marks. The initialization of variable can be done as shown below,

```
struct student  
{  
    char name [5];  
    int roll_number;  
    float avg;  
} s1= {"Ravi", 10, 67.8};
```

The various members of the structure have the following values.

←--- name ----->					← roll_number ->		←----- avg ----->			
R	a	v	i	\0	1	0	6	7	.	8

Figure 5.2 Initial Value of S1

Initialization during Structure declaration:-

Consider the structure definition for student with three fields name, roll number and average marks. The initialization of variable can be done as shown below,

```
typedef struct  
{  
    int x;  
    int y;  
    float t;  
    char u;  
} SAMPLE;  
}
```

SAMPLE sam1 = { 2, 5, 3.2, 'A' };



SAMPLE sam2 = { 7, 3 };

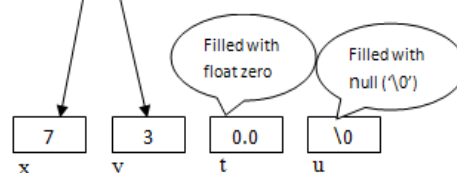


Figure Initializing Structures

The figure shows two examples of structure in sequence. The first example demonstrates what happens when not all fields are initialized. As we saw with arrays, when one or more initializers are missing, the structure elements will be assigned null values, zero for integers and floating-point numbers, and null (“\0”) for characters and strings.

Access the data for structure variables using member operator(.)

We know that variables can be accessed and manipulated using expressions and operators. On the similar lines, the structure members can be accessed and manipulated. The members of a structure can be accessed by using dot(.) operator.

dot (.) operator

Structures use a **dot (.) operator**(also called **period operator** or **member operator**) to refer its elements. Before dot, there must always be a structure variable. After the dot, there must always be a structure element.

The **syntax** to access the structure members as follows,

structure_variable_name . structure_member_name

Consider the example as shown below,

```
struct student
{
    char name [5];
    int roll_number;
    float avg;
};

struct student s1= {"Ravi", 10, 67.8};
```

The members can be accessed using the variables as shown below,

```
s1.name --> refers the string "ravi"
s1.roll_number --> refers the roll_number 10
s1.avg--> refers avg 67.8
```

- 1. Define a structure type *personal*, that would contain person name, date of joining and salary. Write a program to initialize one person data and display the same.**

```
struct personal
{
    char name[20];
    int day;
```

```

char month[10];
int year;
float salary;
};

void main( )
{
struct personal person = { "RAMU", 10 JUNE 1998 20000};
printf("Output values are:\n");
printf("%s%d%s%d%f",person.name,person.day,person.month,person.year,person.salary
);
}

```

Output:- RAMU 10 JUNE 1998 20000

- 2. Define a structure type *book*, that would contain book name, author, pages and price. Write a program to read this data using member operator (.) and display the same.**

```

struct book
{
char name[20];
int day;
char month[10];
int year;
float salary;
};

void main( )
{
struct book b1;
printf("Input values are:\n");
scanf("%s%s%d%f",b1.title, b1.author,b1.pages,b1.price);
printf("Output values are:\n");
printf("%s\n %s\n %d\n %f\n", b1.title, b1.author,b1.pages,b1.price);
}

```

Output:-

Input values are: C& DATA STRUCTURES	KAMTHANE	609350.00
--------------------------------------	----------	-----------

Output values are:

C&	DATA	STRUCTURES
KAMTHANE		
609		
350.00		

NESTED STRUCTURES :

- Structure written inside another structure is called as nesting of two structures.
- Nested Structures are allowed in C Programming Language.
- We can write one Structure inside another structure as member of another structure.

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

struct address
{
    char city[20];
    int pin;
    char phone[14];
};

struct employee
{
    char name[20];
    struct address add;
};

void main ()
{
    struct employee emp;
    printf("Enter employee information?\n");
    scanf("%s %s %d %s",emp.name,emp.add.city, &emp.add.pin, emp.add.phone);
    printf("Printing the employee information....\n");
    printf("name:%s\nCity:%s\nPincode:%d\nPhone:%s",emp.name,emp.add.city,emp.add.pin,emp.add.phone);
}
```

Output:

```
Enter employee information?
Arun
Delhi
110001
1234567890

Printing the employee information....
name: Arun
City: Delhi
Pincode: 110001
Phone: 1234567890
```

ARRAY OF STRUCTURE:

An array is a collection of elements of same data type that are stored in contiguous memory locations. A structure is a collection of members of different data types stored in contiguous memory locations. An array of structures is an array in which each element is a structure. This concept is very helpful in representing multiple records of a file, where each record is a collection of dissimilar data items.

As we have an array of integers, we can have an array of structures also. For example, suppose we want to store the information of class of students, consisting of name, roll_number and marks, A better approach would be to use an array of structures.

Array of structures can be declared as follows,

```
struct tag_name arrayofstructure[size];
```

Let's take an example, to store the information of 3 students; we can have the following structure definition and declaration,

```
struct student
{
char name[10];
int rno;
float avg;
};
struct student s[3];
```

Defines an array called s, which contains three elements. Each element is defined to be of type struct student.

For the student details, array of structures can be initialized as follows,

```
struct student s[3]={{"ABC",1,56.7},{"xyz",2,65.8},{"pqr",3,82.4}};
```

Ex: An array of structures for structure employee can be declared as

```
struct employee emp[5];
```

Let's take an example, to store the information of 5 employees, we can have the following structure definition and declaration,

```
struct employee
{
    int empid;
    char name[10];
    float salary;
};
```

```
struct employee emp[5];
```

Defines array called emp, which contains five elements. Each element is defined to be of type struct employee.

For the employee details, array of structures can be initialized as follows,

```
struct employee emp[5] = {    {1,"ramu",25,20000},
                              {2,"ravi",65000},
                              {3,"tarun",82000},
                              {4,"rupa",5000},
                              {5,"deepa",27000}
};
```

Write a C program to calculate student-wise total marks for three students using array of structure.

```
#include<stdio.h>
struct student
{
    char rollno[10];
    char name[20];
    float sub[3];
    float total;
};

void main( )
{
    int i, j, total = 0;
    struct student s[3];
```



```

printf("\t\t\t Enter 3 students details");

for(i=0; i<3; i++)
{
printf("\n Enter Roll number of %d Student:",i);

gets(s[i].rollno);
printf(" Enter the name:");
gets(s[i].name);
printf(" Enter 3 subjects marks of each student:");
total=0;
for(j=0; j<3; j++)
{
scanf("%d",&s[i].sub[j]);
total = total + s[i].sub[j];
}

}
printf("\n*****");
printf("\n\t\t\t Studentdetails:");
printf("\n*****");
for(i=0; i<n; i++)
{
printf ("\n Student %d:",i+1);
printf("\nRollnumber:%s\n Name:%s",s[i].rollno,s[i].name);
printf ("\nTotal marks =%f", s[i].total);
}
}

```

Practice ☺

1. Write a C program using array of structure to create employee records with the following fields: emp-id, name, designation, address, salary and display it.
2. Write a C program using structure to create a library catalogue with the following fields: Access number, author's name, Title of the book, year of publication, publisher's name, and price.

UNIONS

A union is one of the derived data types. Union is a collection of variables referred under a single name. The syntax, declaration and use of union is similar to the structure but its functionality is different.

The general format or syntax of a union definition is as follows,

Syntax:

```
union union_name
{
<data-type> element 1;
<data-type> element 2;
.....
}union_variable;
```

Example:

```
union techno
{
int comp_id;
char nm;
float sal;
}tch;
```

A union variable can be declared in the same way as structure variable.

union tag_name var1, var2...;

A union definition and variable declaration can be done by using any one of the following

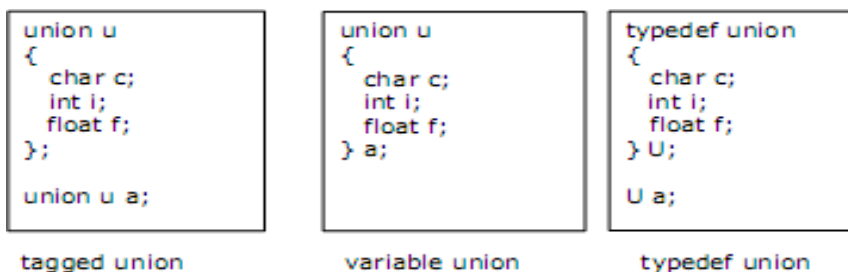


Figure 5.7 Types of Union Definitions

We can access various members of the union as mentioned: a.c a.i a.f and memory organization is shown below,

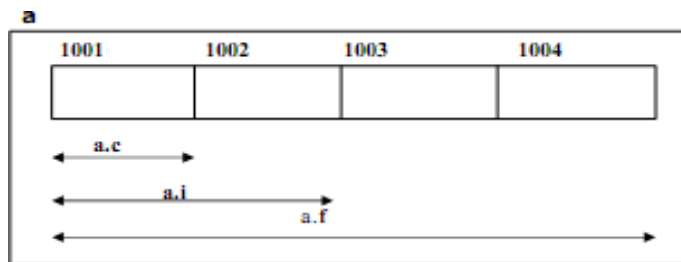


Figure 5.8: Memory Organization Union

In the above declaration, the member **f** requires 4 bytes which is the largest among all the members. Figure 5.8 shows how all the three variables share the same address. The size of the union here is 4 bytes.

A union creates a storage location that can be used by any one of its members at a time. When a different member is assigned a new value, the new value supersedes the previous members' value.

Difference between structure and union:-

	Structure	Union
(i) Keyword	Struct	Union
(ii) Definition	A structure is a collection of logically related elements, possibly of different types, having a single name.	A union is a collection of logically related elements, possibly of different types, having a single name, shares single memory location.
(iii) Declaration	<pre>struct tag_name { type1 member1; type1 member2; }; struct tag_name var;</pre>	<pre>union tag_name { type1 member1; type1 member2; }; union tag_name var;</pre>
(iv) Initialization	Same.	Same.
(v) Accessing	Accessed by specifying structure_variable_name.member_name	Accessed by specifying union_variable_name.member_name

(vi)Memory Allocation	Each member of the structure occupies unique location, stored in contiguous locations.	Memory is allocated by considering the size of the largest member. All the members share the common location
	We can have arrays as a member of structures. All members can be accessed at a time.	We can have array as a member of union. Only one member can be accessed at a time.
	Nesting of structures is possible.	same.
	It is possible structure may contain union as a member.	It is possible union may contain structure as a member

POINTERS:

Pointer is a user defined data type that creates special types of variables which can hold the address of primitive data type like char, int, float, double or user defined data type like function, pointer etc. or derived data type like array, structure, union, enum.

Examples:

```
int *ptr;
```

In c programming every variable keeps two types of value.

1. Value of variable.
2. Address of variable where it has stored in the memory.

(1) Meaning of following simple pointer declaration and definition:

```
int a=5;
int* ptr;
ptr=&a;
```

Explanation:

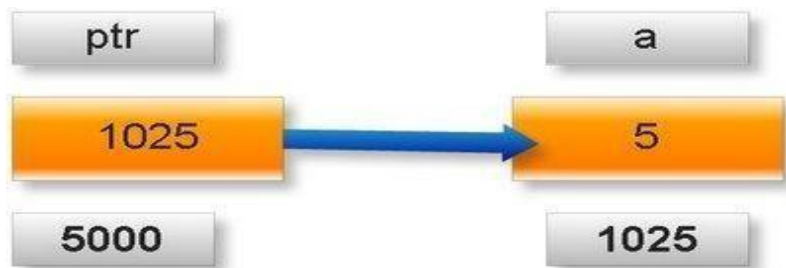
About variable —a :

1. Name of variable: a
2. Value of variable which it keeps: 5
3. Address where it has stored in memory: 1025 (assume)

About variable —ptr :

4. Name of variable: ptr
5. Value of variable which it keeps: 1025
6. Address where it has stored in memory: 5000 (assume)

Pictorial representation:



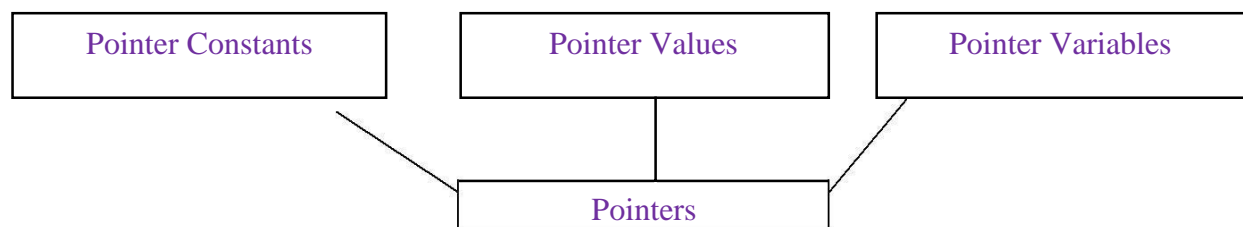
Note: A variable where it will be stored in memory is decided by operating system. We cannot guess at which location a particular variable will be stored in memory.

Pointers are built on three underlying concepts which are illustrated below:-

Memory addresses within a computer are referred to as **pointer constants**. We cannot change them. We can only use them to store data values. They are like house numbers.

We cannot save the value of a memory address directly. We can only obtain the value through the variable stored there using the address operator (&). The value thus obtained is known as **pointer value**. The pointer value (i.e. the address of a variable) may change from one run of the program to another.

Once we have a pointer value, it can be stored into another variable. The variable that contains a pointer value is called a **pointer variable**.



Benefits of using pointers are:-

- 1) Pointers are more efficient in handling arrays and data tables.
- 2) Pointers can be used to return multiple values from a function via function arguments.
- 3) The use of pointer arrays to character strings results in saving of data storage space in memory.
- 4) Pointers allow C to support dynamic memory management.
- 5) Pointers provide an efficient tool for manipulating dynamic data structures such as structures , linked lists , queues , stacks and trees.
- 6) Pointers reduce length and complexity of programs.
- 7) They increase the execution speed and thus reduce the program execution time.

In C, every variable must be declared for its type. Since pointer variable contain addresses that belong to a separate data type ,they must be declared as pointers before we use them.

a) Declaration of a pointer variable:

The declaration of a pointer variable takes the following form:

data_type *pt_name;

This tells the compiler three things about the variable pt_name:

- 1) The * tells that the variable pt_name is a pointer variable
- 2) pt_name needs a memory location
- 3) pt_name points to a variable of type data_type

Ex: int *p;

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

b) Initialization of pointer variables:

The process of assigning the address of a variable to a pointer variable is known as **initialization**. Once a pointer variable has been declared we can use assignment operator to initialize the variable.

Ex:

```
int quantity ;  
int *p;          //declaration  
p=&quantity;     //initialization
```

We can also combine the initialization with the declaration:

```
int *p=&quantity;
```

Always ensure that a pointer variable points to the corresponding type of data.

It is also possible to combine the declaration of data variable, the declaration of pointer variable and the initialization of the pointer variable in one step.

```
int x, *p=&x;
```

& AND * OPERATORS:

The actual location of a variable in the memory is system dependent and therefore the address of a variable is not known to us immediately. In order to determine the address of a variable we use & operator in c. This is also called as the **address operator**. The operator & immediately preceding a variable returns the address of the variable associated with it.

P=&x; would assign the address 5000 to the variable p.

The & operator can be remembered as **address of**.

Example program:

```
main( )
{
int a = 5 ;
printf ( "\nAddress of a = %u", &a
); printf ( "\nValue of a = %d", a );
}
```

Output: The output of the above program would be:

Address of a=1444

value of a=5

The expression &a returns the address of the variable a, which in this case happens to be 1444 .Hence it is printed out using %u, which is a format specified for printing an unsigned integer.

Accessing a variable through its pointer:

Once a pointer has been assigned the address of a variable, to access the value of the variable using pointer we use the operator ‘*’, **called ‘value at address’ operator**. It gives the value stored at a particular address. The „value at address” operator is also called **‘indirection’ operator (or dereferencing operator)**.

Ex:

```
main()
{
int a = 5 ;
printf ( "\nAddress of a = %u", &a );
printf ( "\nValue of a = %d", a ) ;
printf ( "\nValue of a = %d", *( &a ) );

}
```

Output: The output of the above program would be:

Address of a = 1444

Value of a = 5

Value of a = 5

Various arithmetic operations that can be performed on pointers

- Like normal variables, pointer variables can be used in expressions.

Ex: $x = (*p1) + (*p2);$

- C language allows us to add integers to pointers and to subtract integers from pointers

Ex: If $p1, p2$ are two pointer variables then operations such as $p1+4, p2 - 2, p1 - p2$ can be performed.

- Pointers can also be compared using relational operators.

Ex: $p1 > p2, p1 == p2, p1 != p2$ are valid operations.

- We should not use pointer constants in division or multiplication. Also, two pointers cannot be added. $p1/p2, p1*p2, p1/3, p1+p2$ are invalid operations.

- Pointer increments and scale factor:-

Let us assume the address of $p1$ is 1002. After using $p1 = p1 + 1$, the value becomes 1004 but not 1003.

Thus when we increment a pointer, its value is increased by length of data type that points to. This length is called scale factor.

Pointers and Arrays :-

When an array is declared the compiler allocates a base address and sufficient amount of storage to contain all the elements of the array in contiguous memory locations. The base address is the location of the first element (index 0) of the array. The compiler also defines the array name as a constant pointer to the first element.

Ex:- static int x[5]= { 1,2,3,4,5};

Suppose the base address of x is 1000 and assuming that each integer requires two bytes. The five elements will be stored as follows.

elements	x[0]	x[1]	x[2]	x[3]	x[4]
value	1	2	3	4	5
address	1000	1002	1004	1006	1008

the name x is defined as a constant pointer pointing to the first element, x[0], and therefore the value x is 1000, the location where x[0] is stored. That is

$x = \&x[0] = 1000;$

If we declare p as an integer pointer, then we can make the pointer p to the array x by the following assignment

$p = x;$

which is equivalent to $p = \&x[0];$

Now we can access every value of x using p++ to move from one element to another. The relationship between p and x is shown below

$p = \&x[0] = 1000$
 $p+1 = \&x[1] = 1002$
 $p+2 = \&x[2] = 1004$
 $p+3 = \&x[3] = 1006$
 $p+4 = \&x[4] = 1008$

Note:- address of an element in an array is calculated by its index and scale factor of the datatype

Address of x[n] = base address + (n*scale factor of type of x).

Eg:- int x[5]; x=1000;

Address of x[3] = base address of x + (3*scale factor of int)
= $1000 + (3*2)$
= $1000 + 6$

=1006

Ex: - float avg [20];
avg=2000;

Address of avg [6]=2000+(6*scale factor of float) =2000+6*4 =2000+24
=2024.

Ex:- char str [20]; str =2050;
Address of str[10]=2050+(10*1)
=2050+10
=2060.

Note2:- when handling arrays, of using array indexing we can use pointers to access elements.
Like *(p+3) given the value of x[3]

The pointer accessing method is faster than the array indexing.

Accessing elements of an array:-

```
/*Program to access elements of a one dimensional array*/
#include<stdio.h>
#include<conio.h>
void main()
{

int arr[5]={ 10,20,30,40,50};
int p=0;
printf("\n value@ arr[i] is arr[p] | *(arr+p)| *(p+arr) | p[arr] | located @address\n");

for(p=0;p<5;p++)
{
printf("\n value of arr[%d] is:",p);
printf(" %d | ",arr[p]);
printf(" %d | ",*(arr+p));

printf(" %d | ",*(p+arr));

printf(" %d | ",p[arr]);

printf("address of arr[%d]=%u\n",p,arr[p]);
}
}
```

output:

```
value@ arr[i] is arr[p] | *(arr+p) | *(p+arr) | p[arr] | located @address
value of arr[0] is: 10   | 10   | 10   | 10   | address of arr[0]=10
value of arr[1] is: 20   | 20   | 20   | 20   | address of arr[1]=20
value of arr[2] is: 30   | 30   | 30   | 30   | address of arr[2]=30
value of arr[3] is: 40   | 40   | 40   | 40   | address of arr[3]=40
value of arr[4] is: 50   | 50   | 50   | 50   | address of arr[4]=50
-
```

Pointers to two-dimensional arrays:-

Pointer can also be used to manipulate two-dimensional arrays. Just like how $x[i]$ is represented by $*(x+i)$ or $*(p+i)$, similar, any element in a 2-d array can be represented by the pointer as follows.

$*(*(a+i)+j)$ or $*(*(p+i)+j)$

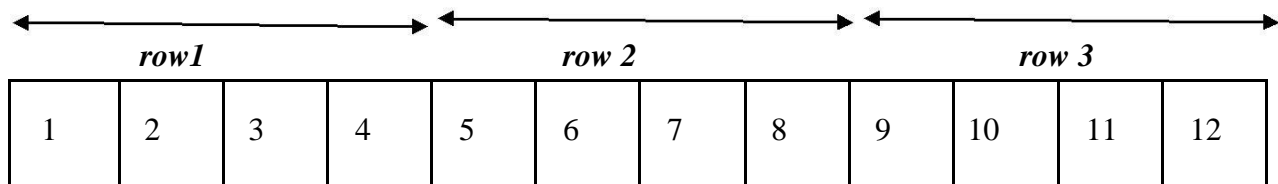
The base address of the array a is $\&a[0][0]$ and starting from this address the compiler allocates contiguous space for all the element row – wise .

i.e the first element of the second row is placed immediately after the last element of the first row and so on

Ex:- suppose we declare an array as follows.

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

The elements of a will be stored as shown below.



Base address = $\&a[0][0]$

If we declare p as int pointer with the initial address $\&a[0][0]$ then $a[i][j]$ is equivalent to $*(p+4 \times i+j)$. You may notice if we increment „ i “ by 1, the p is incremented by 4, the size of each row. Then the element $a[2][3]$ is given by $*(p+2 \times 4+3) = *(p+11)$.

This is the reason why when a two –dimensional array is declared we must specify the size of the each row so that the compiler can determine the correct storage at mapping

Program:

```
/* accessing elements of 2 d array using pointer*/
#include<stdio.h>
void main()
{
int z[3][3]={ {1,2,3},{4,5,6},{7,8,9}};
int i, j;
int *p;
p=&z[0][0];

for(i=0;i<3;i++)
{
for(j=0;j<3;j++)
{
printf("\n %d\ti=%d j=%d\t%d",*(p+i*3+j),i,j,*(z+i+j));
}
}
for(i=0;i<9;i++)
printf("\n\t%d",*(p+i));
}
```

Output:

1	i=0	j=0	1
2	i=0	j=1	2
3	i=0	j=2	3
4	i=1	j=0	4
5	i=1	j=1	5
6	i=1	j=2	6
7	i=2	j=0	7
8	i=2	j=1	8
9	i=2	j=2	9
	1		
	2		
	3		
	4		
	5		
	6		
	7		
	8		
	9		
	_		

Array of Pointers(Pointer arrays):-

We have studied array of different primitive data types such as int, float, char etc. Similarly C supports array of pointers i.e. collection of addresses.

data_type *array_name [exp1];

Example :-

```
void main()
{
int *ap[3];
int al[3]={ 10,20,30};
int k;

for(k=0;k<3;k++)
ap[k]=al+k;
printf("\naddress  element\n");
for(k=0;k<3;k++)
{
printf("\t %u",ap[k]);
printf("\t %7d\n",*(ap[k]));

}
}
```

Output:

Address	Element
4060	10
4062	20
4064	30

In the above program, the addresses of elements are stored in an array and thus it represents array of pointers.

A two-dimensional array can be represented using pointer to an array. But, a two-dimensional array can be expressed in terms of array of pointers also. The conventional array definition is,

data_type array name [exp1] [exp2];

Using array of pointers, a two-dimensional array can be defined as,

data_type *array_name [exp1];

Where,

- data_type refers to the data type of the array.
- array_name is the name of the array.
- exp1 is the maximum number of elements in the row.

Note that exp2 is not used while defining array of pointers. Consider a two-dimensional vector initialized with 3 rows and 4 columns as shown below,

```
int p[3][4]={{ 10,20,30,40},{50,60,70,80},{25,35,45,55}};
```

The elements of the matrix p are stored in memory row-wise (can be stored column-wise also) as shown in Fig.

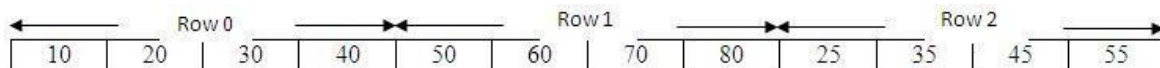


Fig: Two-Dimensional Array

Using array of pointers we can declare p as, `int *p[3];`

Here, p is an array of pointers. p[0] gives the address of the first row, p[1] gives the address of the second row and p[2] gives the address of the third row. Now, p[0]+0 gives the address of the

element in 0th row and 0th column, p[0]+1 gives the address of the elements in 0th row and 1st column and so on. In general,

- Address of ith row is given by a[i].

Address of an item in ith row and jth column is given by, p[i]+j. The element in ith row and jth column can be accessed using the indirection operator * by specifying, *(p[i]+j)

Pointers to structures:-

We have pointers pointing to int, float, arrays etc., We also have pointers pointing to structures. They are called as structure pointers.

To access the members of a structure using pointers we need to perform The following operations:

- Declare the structure variable
- Declare a pointer to a structure
- Assign address of structure variable to pointer
- Access members of structure using (.) operator or using (->)member selection operator or arrow operator.

Syntax:

```
struct tagname
{
    datatype member1;
    datatype member2;
    ...
};
```

```
struct tagname var;
struct tagname *ptr;
ptr=&var;
```

to access the members

(*ptr).member1; or ptr->member1;

The parentheses around *ptr are necessary because the member operator ' . ' has a higher precedence than the operator ' * '

Example:

```
struct student
{
    int rno;
    char name[20];
};

struct student s1;
struct student *ptr;
ptr=&s1;
to access (*ptr).rno;(*ptr).name; (or)ptr->rno; ptr->name;
```

Program to read and display student details using pointers to structures

```
struct student
{
    int HTNO;
    char NAME[20];
    float AVG;
};

void main()
{
    struct student s1;
    struct student *ptr;
    ptr=&s1;
    printf("Enter student details");
    scanf("%d%s%f",&ptr->HTNO,ptr->NAME,&ptr->AVG);
    printf("HTNO=%d",ptr->HTNO);
    printf("NAME=%s",ptr->NAME);
    printf("AVERAGE MARKS=%f",ptr->AVG);
}
```

Self-referential structure

A structure definition which includes at least one member as a pointer to the same structure is known as self-referential structure.

It can be linked together to form useful data structures such as lists, queues, stacks and trees. It is terminated with a NULL pointer.

The syntax for using the self referential structure is as follows,

```
struct tag_name
{
    Type1 member1; Type2 member2;
    .....
    struct tag_name *next;
};
```

Ex:-

```
struct node
{
    int data;
    struct node *next;
} n1, n2;
```


ENUMERATED DATA TYPE:

An enumeration is a user-defined data type consists of integral constants and each integral constant is given a name. Keyword **enum** is used to define enumerated data type.

```
enum type_name{ value1, value2,...,valueN };
```

Here, *type_name* is the name of enumerated data type or tag. And *value1,value2,...,valueN* are values of type *type_name*.

By default, *value1* will be equal to 0, *value2* will be 1 and so on but, the programmer can change the default value.

```
// Changing the default value of enum elements
```

```
enum suit {club=0; diamonds=10; hearts=20; spades=3};
```

Declaration of enumerated variable

Above code defines the type of the data but, no any variable is created. Variable of type **enum** can be created as:

```
enum boolean{  
    false;  
    true;  
};
```

```
enum boolean check;
```

Here, a variable *check* is declared which is of type **enum boolean**.

Example of enumerated type

```
#include <stdio.h>  
enum week{ sunday, monday, tuesday, wednesday, thursday, friday, saturday};  
  
int main(){  
    enum week today;  
    today=wednesday;  
    printf("%d day",today+1);  
    return 0;  
}
```

Output

4 day

You can write any program in C language without the help of enumerations but, enumerations helps in writing clear codes and simplify programming.

SHORT ANSWERS ☺

1. Why is it necessary to give the size of an array in array declaration? [May 2019]
2. Mention the advantages and disadvantages of array [May 2019]
3. How pointer can be used for accessing multi dimensional arrays? Discuss [Dec 2018]
4. How to declare a string without specifying length [Aug 2019]
5. Discuss about strlen() function with example [Dec 2018]
6. Give syntax to create a pointer to function [June 2019]
7. Write the advantages and disadvantages of using pointers [June 2019]
8. Give a note on unions [June 2019]
9. What is an enumerated data type and write syntax and example [Aug 2019]
10. What are string functions? Explain.
11. Define structure with example.
12. Difference between structure and union.
13. Give an example for nested structure.
14. What is self referential structure? Explain.
15. What is the purpose of pointers?
16. Give an example for pointers to pointers.
17. Convert the size of pointer to different data types (int*, float*, float*)
18. Explain array of pointers with example.
19. What is null pointer? What is a void pointer? Explain when null pointer and void pointers are used.
20. Explain how does enum differ from type def in C.

LONG ANSWERS ☺

ARRAYS

1. Write down the applications of using array. [may 2019]
2. Define an array. What are the different types of arrays? Explain
3. Write the syntax for declaring two - dimensional array write a program to access and print the array elements
4. Write the program to find the sum of even numbers using arrays
5. Write a 'C' program to find the biggest number and smallest number of given 'n' numbers using arrays [June 2019]
6. Explain multidimensional arrays and give an example to pass arrays as argument in functions. [aug 2019]
7. Consider the array declaration float a [5]; and the memory address of a [0] is 4056, what is the memory address of a [2]? [Dec 2018]
8. What is multidimensional array? Explain how a multidimensional array is defined in terms of a pointer to a collection of contiguous arrays of lower dimensionality. [may 2019]
9. Explain how two dimensional arrays can be used to represent matrices. Write C code to perform matrix addition and matrix multiplication
10. Using arrays and iteration, Write C-language program that outputs minimum number of currency notes required for the given amount at input. For example, an amount of Rs.4260 shall output 1000s – 4; 100s – 2; 50s -1; 10s-1. The currency denominations are 1000, 500, 100, 50,20,10,5,2 and 1.

STRINGS *

1. Discuss any five string handling functions in detail. [Aug 2019]
2. Write a program to find the string length by using string functions. [Dec 2018]
3. Write a complete C program that reads a string and prints if it is a palindrome or not.

STRUCTURES

1. What is a structure? Explain how to declare, initialize and access the structure elements. [Dec 2018]
2. Write and explain the general format for declaring and accessing members of a structure. [may 2019]
3. Why structures are necessary? Explain nested structures with valid example. [June 2019]
4. How to pass the structures to functions as argument? Explain. (Aug 2019)
5. Difference between structure and union in C. [may 2019]
6. Write a program to declare pointer to a structure and display the contents of the structure. Define a structure object of book with three fields: title, author and pages.
7. Explain the following with suitable examples. i. typedef ii. Self referential structures
8. How is an array of structures initialized?
9. Explain the following with examples: a) Nested structures b) Array of structures c) Unions.
10. How is structure different from an array? Explain.

11. Explain how complex numbers can be represented using structures. Write two C functions: one to return the sum of two complex numbers passed as parameters, and another to return the product of two complex numbers passed as parameters
12. Write C-structures for the College data. College contains the following fields: College code (2characters), College Name (dynamically allocated string), year of establishment, number of courses and courses(dynamically allocated structure). A College can offer 1 to 50 courses. Each course is associated with course name (String), duration, number of students. The number of students in the college is sum of number of students in all the courses in the college. Write a function `int collegeStrength (struct College *c)` that returns the number of students in the college pointed by c.
13. Write C-structures for a country with the following fields: country Name (dynamically allocated string), currency code (3 letter string), number of states and states (dynamically allocated structures). A country can have 1 to 100 states. Each state is associated with name (dynamically allocated string), area, and population. The area of the country is the sum of the areas of all the states in the country. Similarly the population of the country is population of all states put together. Write function `void countrystats(struct country *c, int *a, int *b)` that computes the area and population of the country and places at locations pointed by a and b.
14. Write C-structures for departmental store application. Each departmental store contains departmental store Id (3 characters), store location (dynamically allocated string), items (dynamically allocated structures) and number of items. A store can offer 1 to 1000 items. Each Item contains Item code (4 characters), current stock, unit of measure in the following set (Single, dozen, kilogram, liter, meter, square meter) and price. Using this structure, Write C- function to count the number of items whose price is above the given amount.
15. How to copy one structure to another structure of a same data type, give an example?

POINTERS

1. Give a detailed note on pointer expressions. [June 2019]
2. How to use pointer as arguments in a function? Explain with a program. [May 2019]
3. Write a C program to illustrate the use of array of pointers.
4. List out the advantages of using pointers and explain generic (void) pointers with suitable example.
5. Write and explain the program for pointer to multi dimensional arrays.

ENUMERATED DATATYPES *

1. Give a brief note on Enumerated data types. [June 2019]
2. Explain about Enumerated data types with an example. [Dec 2018]

UNIT-III

PREPROCESSOR AND FILE HANDLING IN C

Preprocessor directives

Before a C program is compiled in a compiler, source code is processed by a program called preprocessor. This process is called preprocessing.

Commands used in preprocessor are called preprocessor directives and they begin with “#” symbol. Below is the list of preprocessor directives that C language offers.

S.no	Preprocessor	Syntax	Description
1	Macro	#define	This macro defines constant value and can be any of the basic data types.
2	Header file inclusion	#include<file_name>	The source code of the file “file_name” is included in the main program at the specified place.
3	Conditional compilation	#ifdef, #endif, #if, #else, #ifndef	Set of commands are included or excluded in source program before compilation with respect to the condition.
4	Other Directives	#undef, #pragma	#undef is used to undefine a defined macro variable. #Pragma is used to call a function before and after main function in a C program.

EXAMPLE PROGRAM FOR #DEFINE, #INCLUDE PREPROCESSORS IN C:

#define – This macro defines constant value and can be any of the basic data types.

#include <file_name> – The source code of the file “file_name” is included in the main C program where “#include <file_name>” is mentioned.

```
#include <stdio.h>
#define height 100
#define number 3.14
#define letter 'A'
#define letter_sequence "ABC"
#define backslash_char '\\'

void main()
{
    printf("value of height : %d \n", height );
    printf("value of number : %f \n", number );
    printf("value of letter : %c \n", letter );
    printf("value of letter_sequence : %s \n", letter_sequence);
    printf("value of backslash_char : %c \n", backslash_char);
}
```

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

EXAMPLE PROGRAM FOR CONDITIONAL COMPILATION DIRECTIVES:

A) EXAMPLE PROGRAM FOR #IFDEF, #ELSE AND #ENDIF IN C:

“#ifdef” directive checks whether particular macro is defined or not. If it is defined, “if” clause statements are included in source file. Otherwise, “else” clause statements are included in source file for compilation and execution.

```
#include <stdio.h>
#define RAJU 100

int main()
{
    #ifdef RAJU
    printf("RAJU is defined. So, this line will be added in " \ "this C file\n");
    #else
    printf("RAJU is not defined\n");
    #endif
    return 0;
}
```

```
RAJU is defined. So, this line will be added in this C file
```

B) EXAMPLE PROGRAM FOR #IFNDEF AND #ENDIF IN C:

#ifndef exactly acts as reverse as #ifdef directive. If particular macro is not defined, “if” clause statements are included in source file. Otherwise, else clause statements are included in source file for compilation and execution.

```
#include <stdio.h>
#define RAJU 100

int main()
{
    #ifndef SELVA
    {
        printf("SELVA is not defined. So, now we are going to " \ "define here\n");
    }
}
```

```

#define SELVA 300
}
#else
printf("SELVA is already defined in the program");

#endif
return 0;
}

```

SELVA is not defined. So, now we are going to define here

C) EXAMPLE PROGRAM FOR #IF, #ELSE AND #ENDIF IN C:

“If” clause statement is included in source file if given condition is true.

Otherwise, else clause statement is included in source file for compilation and execution.

```

#include <stdio.h>
#define a 100
int main()
{
    #if (a==100)
    printf("This line will be added in this C file since " \ "a \= 100\n");
    #else
    printf("This line will be added in this C file since " \ "a is not equal to 100\n");
    #endif
    return 0;
}

```

This line will be added in this C file since a = 100

EXAMPLE PROGRAM FOR UNDEF IN C:

This directive undefines existing macro in the program.

```

#include <stdio.h>

#define height 100
void main()
{
    printf("First defined value for height   : %d\n",height);
    #undef height      // undefining variable
    #define height 600  // redefining the same for new value
    printf("value of height after undef \& redefine:%d",height);
}

```

First defined value for height : 100
value of height after undef & redefine : 600

EXAMPLE PROGRAM FOR PRAGMA IN C: (Shall be discussed after the unit-IV functions)

Pragma is used to call a function before and after main function in a C program.

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

```
void function1( );
```

```
void function2( );
```

```
#pragma startup function1
```

```
#pragma exit function2
```

```
int main( )
```

```
{
```

```
    printf ( "\n Now we are in main function" );
```

```
    return 0;
```

```
}
```

```
void function1( )
```

```
{
```

```
    printf("\nFunction1 is called before main function call");
```

```
}
```

```
void function2( )
```

```
{
```

```
    printf ( "\nFunction2 is called just before end of " \ "main function" );"
```

```
}
```

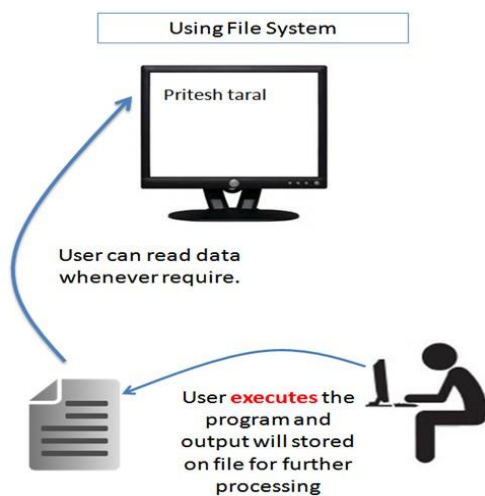
Function1 is called before main function call
Now we are in main function
Function2 is called just before end of main function

MORE ON PRAGMA DIRECTIVE IN C:

S.no	Pragma command	description
1	#Pragma startup <function_name_1>	This directive executes function named “function_name_1” before
2	#Pragma exit <function_name_2>	This directive executes function named “function_name_2” just before termination of the program.
3	#pragma warn – rv1	If function doesn’t return a value, then warnings are suppressed by this directive while compiling.
4	#pragma warn – par	If function doesn’t use passed function parameter , then warnings are suppressed
5	#pragma warn – rch	If a non reachable code is written inside a program, such warnings are suppressed by this directive.

FILE HANDLING IN C

- ❖ A file is an external collection of related data treated as a unit. A file is a place on a disk where a group of related data is stored and retrieved whenever necessary without destroying data.
- ❖ The primary purpose of a file is to keep record of data. Record is a group of related fields. Field is a group of characters which convey meaning.
- ❖ Files are stored in auxiliary or secondary storage devices. The two common forms of secondary storage are disks (hard disk, CD and DVD) and tapes.
- ❖ Each file ends with an end of file (EOF) at a specified byte number, recorded in file structure.
- ❖ A file must first be opened properly before it can be accessed for reading or writing. When a file is opened an object (buffer) is created and a stream is associated with the object.



It is advantageous to use files in the following circumstances.

1. When large volume of data are handled by the program and
2. When the data need to be stored permanently without getting destroyed when program is terminated.

There are two kinds of files depending upon the format in which data is stored:

- 1) Text files
- 2) Binary files

1) Text files:

A text file stores textual information like alphabets, numbers, special symbols, etc. actually the ASCII code of textual characters its stored in text files. Examples of some text files include c, java, c++ source code files and files with .txt extensions. The text file contains the characters in sequence. The computer process the text files sequentially and in forward direction. One can perform file reading, writing and appending operations. These operations are performed with the help of inbuilt functions of c.

2) Binary files:

Text mode is inefficient for storing large amount of numerical data because it occupies large space. Only solution to this is to open a file in binary mode, which takes lesser space than the text mode. These files contain the set of bytes which stores the information in binary form. One main drawback of binary files is data is stored in human unreadable form. Examples of binary files are .exe files, video stream files, image files etc. C language supports binary file operations with the help of various inbuilt functions.

To store data in a file three things have to be specified for operating system. They include

1. FILENAME:

It is a string of characters that make up a valid file name which may contain two parts, a primary name and an optional period with the extension.

Prog1.c Myfirst.java Data.txt store

2. DATA STRUCTURE:

It is defined as FILE in the library of standard I/O function definitions. Therefore all files are declared as type FILE. FILE is a define data type.

FILE *fp;

3. PURPOSE:

It defines the reason for which a file is opened and the mode does this job.

fp=fopen("filename", "mode");

Basic operations on file:

1. Naming a file

2. Opening a file

3. Reading data from file

4. Writing data into file

5. Closing a File

In order to perform the basic file operations C supports a number of functions .Some of the important file handling functions available in the C library are as follows:

FUNCTION NAME	OPERATION
fopen()	Creates a new file for use
	Opens an existing file for use
fclose()	Closes a file which has been opened for use
fcloseall()	Closes all files which are opened
getc()/fgetc()	Reads a character from a file
putc()/fputc()	Writes a character to a file

gets()	Reads a string from a file
puts()	Writes a string to a file
fprintf()	Writes a set of data values to files
fscanf()	Reads a set of data values from files
getw()	Reads an integer from file
putw()	Writes an integer to a file
ftell()	Gives the current position in the file
fseek()	Sets the position to a desired point in a file
rewind()	Sets the position to the beginning of the file

Naming and opening a file:

A name is given to the file used to store data. The file name is a string of characters that make up a valid file name for operating system. It contains two parts. A primary name and an optional period with the extension.

Examples: Student.dat, file1.txt, marks.doc, palin.c.

The general format of declaring and opening a file is

FILE *fp;//declaration

fp=fopen ("filename","mode");//statement to open file.

Here FILE is a data structure defined for files. fp is a pointer to data type FILE. filename is the name of the file. mode tells the purpose of opening this file.

THE DIFFERENT MODES OF OPENING FILES:

"r" (read) mode:

The read mode (r) opens an existing file for reading. When a file is opened in this mode, the file marker or pointer is positioned at the beginning of the file (first character). The file must already exist: if it does not exist a NULL is returned as an error. If we try to write a file opened in read mode, an error occurs.

Syntax: **fp=fopen ("filename","r");**

"w" (write) mode:

The write mode (w) opens a file for writing. If the file doesn't exist, it is created. If it already exists, it is opened and all its data is erased; the file pointer is positioned at the beginning of the file. It gives an error if we try to read from a file opened in write mode.

Syntax: **fp=fopen ("filename","w");**

“a” (append) mode:

The append mode (a) opens an existing file for writing instead of creating a new file. However, the writing starts after the last character in the existing file that is new data is added, or appended, at the end of the file. If the file doesn't exist, new file is created and opened. In this case, the writing will start at the beginning of the file.

Syntax: `fp=fopen (“filename”,“a”);`

“r+” (read and write) mode:

In this mode file is opened for both reading and writing the data. If a file does not exist then NULL, is returned.

Syntax: `fp=fopen (“filename”,“r+”);`

“w+” (read and write) mode:

In this mode file is opened for both writing and reading the data. If a file already exists its contents are erased and if a file does not exist then a new file is created.

Syntax: `fp=fopen (“filename”,“w+”);`

“a+” (append and read) mode:

In this mode file is opened for reading the data as well as data can be added at the end.

Syntax: `fp=fopen (“filename”, “a+”);`

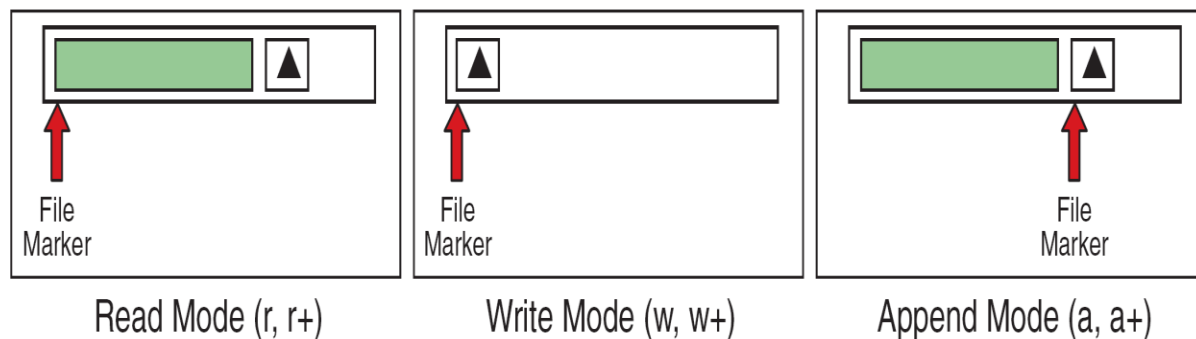


Figure : File Opening Modes

NOTE: To perform operations on binary files the following modes are applicable with an extension b like rb,wb,ab,r+b,w+b,a+b, which has the same meaning but allows to perform operations on binary files.

Reading data from file :

Input functions used are (Input operations on files)

- a) **getc();** It is used to read characters from file that has been opened for read operation.

Syntax: c=getc (file pointer)

This statement reads a character from file pointed to by file pointer and assign to c.

It returns an end-of-file marker EOF, when end of file has been reached.

- b) **fscanf();**

This function is similar to that of scanf function except that it works on files.

Syntax: fscanf (fp, "control string", list);

Example fscanf(f1, "%s%d", str, &num);

The above statement reads string type data and integer type data from file.

- c) **getw();** This function reads an integer from file.

Syntax: getw (file pointer);

- d) **fgets();** This function reads a string from a file pointed by a file pointer. It also copies the string to a memory location referred by an array.

Syntax: fgets(string, no of bytes, filepointer);

- e) **fread();** This function is used for reading an entire structure block from a given file.

Syntax: fread(&struct_name, sizeof(struct_name), 1, filepointer);

Writing data to a file:

To write into a file, following C functions are used

- a) **putc();** This function writes a character to a file that has been opened in write mode.

Syntax: putc(c, fp);

This statement writes the character contained in character variable c into a file whose pointer is fp.

- b) **fprintf();** This function performs function, similar to that of print except that it works on files.

Syntax: fprintf(f1, "%s,%d", str, num);

- c) **putw();** It writes an integer to a file.

Syntax: putw (variable, fp);

- d) **fputs();** This function writes a string into a file pointed by a file pointer.

Syntax: fputs (string, filepointer);

- e) **fwrite();** This function is used for writing an entire block structure to a given file.

Syntax: fwrite(&struct_name, sizeof(struct_name), 1, filepointer);

Closing a file:

A file is closed as soon as all operations on it have been completed. Closing a file ensures that all outstanding information associated with the file is flushed out from the buffers and all links to the file are broken. Another instance where we have to close a file is to reopen the same file in a different mode. Library function for closing a file is

fclose(file pointer);

Example: fclose(fp);

Where fp is the file pointer returned by the call to fopen(). fclose() returns 0 on success (or) - 1 on error. Once a file is closed, its file pointer can be reused for another file.

Note: fcloseall() can be used to close all the opened files at once.

What are the file I/O functions in C. Give a brief note about the task performed by each function.

Ans: In order to perform the file operations in C we must use the high level I/O functions which are in C standard I/O library. They are

i) getc() and putc() functions:

getc()/fgetc() : It is used to read a character from a file that has been opened in a read mode. It reads a character from the file whose file pointer is fp. The file pointer moves by one character for every operation of getc(). The getc() will return an end-of –marker EOF, when an end of file has been reached.

Syntax: **getc(fp);**

Ex: char ch;

ch=getc(fp);

putc()/fputc() -: It is used to write a character contained in the character variable to the file associated with the FILE pointer fp. fputc() also returns an end-of –marker EOF, when an end of file has been reached.

Syntax: **putc(c,fp);**

Example: char c;

putc(c,fp);

Program using fgetc() and fputc():

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

```
void main()
```

```
{
```

```
FILE *fp;
```

```
char ch;
```

```

fp=fopen("input1.txt","w");

if(fp== NULL)
{
printf("\n Error opening a file");
exit(0);
}

printf("\n enter some text here nd press cntrl D or Z to stop :\n");

while((ch=getchar())!=EOF)
fputc(ch,fp);
fclose(fp);

fp=fopen("input1.txt","r");
printf("\n The entered text is : \n");
while((ch=fgetc(fp))!=EOF)
putchar(ch);
fclose(fp);
}

```

ii) fprintf() and fscanf():

In order to handle a group of mixed data simultaneously there are two functions that are fprintf() and fscanf(). These two functions are identical to printf and scanf functions, except that they work on files. The first argument of these functions is a file pointer which specifies the file to be used.

fprintf():

The general form of fprintf() is

Syntax: **fprintf(fp,"control string",list);**

where fp is a file pointer associated with a file that has been opened for writing . The control string contains output specifications for the items in the list. .

Example: **fprintf(fp,"%s%d",name,age);**

fscanf() : It is used to read a number of values from a file.

Syntax: **fscanf(fp,"control string",list);**

Example: **fscanf(fp2,"%s %d",item,&quantity);**

like scanf , fscanf also returns the number of items that are successfully read. when the end of file is reached it returns the value EOF.

Program using fscanf() and fprintf():

```
#include<stdio.h>
void main()
{
    int a=22,b;
    char s1[20]="Welocme_to_c",s2[20];
    float c=12.34,d;
    FILE *f3;
    f3=fopen("mynew3.txt","w");
    fprintf(f3,"%d  %s  %f",a,s1,c);
    fclose(f3);
    f3=fopen("mynew3.txt","r");
    fscanf(f3,"%d %s %f",&b,s2,&d);
    printf("\n a=%d \t s1=%s \t c=%f \n b=%d \t s2=%s \t d=%f",a,s1,c,b,s2,d);
    fclose(f3);
}
```

iii) getw() and putw():

The getw() and putw() are integer oriented functions. They are similar to the getc() and putc() functions and are used to read and write integer values. These functions would be useful when we deal with only integer data. The general form of getw() and putw() are

Syntax: **putw(integer,fp);**

Syntax: **getw(fp);**

Program using getw() and putw():

```
/*Printing odd numbers in odd file and even numbers in even file*/
#include<stdio.h>
void main()
{
    int x,i;
    FILE *f1,*fo,*fe;     //creating a file pointer
    f1=fopen("num.txt","w");     //opening a file
    printf("\n enter some numbers into file or -1 to stop \n");
    for(i=0;i<20;i++)
    {
        scanf("%d",&x);
        if(x== -1)
```

```

break;
putw(x,f1); //writing read number into anil.txt file one at a time
}

fclose(f1); //closing a file opened for writing input
printf("\nOUTPUTDATA\n");
f1=fopen("num.txt","r");//open anil in read mode to read data
fo=fopen("odd.txt","w");
fe=fopen("even.txt","w");

while((x=getw(f1))!=EOF)
{
printf("%d\t",x);
if(x%2==0)
putw(x,fe);
else
putw(x,fo);
}

fcloseall();
fo=fopen("odd.txt","r");
printf("\n contents of odd file are :\n");
while((x=getw(fo))!= EOF)
printf(" %d\t",x);
fe=fopen("even.txt","r");
printf("\n contents of even file are :\n");
while((x=getw(fe)) != EOF)
printf(" %d\t",x);
fcloseall();
}

```

iv) **fputs()** and **fgets()**:

fgets(): It is used to read a string from a file pointed by file pointer. It copies the string to a memory location referred by an array.

Syntax: **fgets(string,length,filepointer);**

Example: **fgets(text,50,fp1);**

fputs(): It is used to write a string to an opened file pointed by file pointer.

Syntax: **fputs(string,filepointer);**

Example: **fputs(text,fp);**

Program using fgets() and fputs():

```
#include<stdio.h>
void main()
{
    FILE *fp; char str[50];
    fp=fopen("fputget.txt","r");
    printf("\n the read string is :\n");
    fgets(str,50,fp);
    puts(str);
    fclose(fp);
    fp=fopen("fputget.txt","a+");
    printf("\n Enter string : \n");
    gets(str);
    fputs(str,fp);
    puts(str);
    fclose(fp);
}
```

Block or structures read and write:

Large amount of integers or float data require large space on disk in text mode and turns out to be inefficient .For this we prefer binary mode and the functions used are

v) fread() and fwrite():

fwrite(): It is used for writing an entire structure block to a given file in binary mode.

Syntax: **fwrite(&structure_variable,sizeof(structure_variable),1,filepointer);**

Example: **fwrite(&stud,sizeof(stud),1,fp);**

fread(): It is used for reading an entire structure block from a given file in binary mode.

Syntax: **fread(&structure_variable,sizeof(structure_variable),1,filepointer);**

Example: **fread(&emp,sizeof(emp),1,fp1);**

Program using fread() and fwrite():

```
#include<stdio.h>
struct player
{
    char pname[30];
    int age;
    int runs;
};
```

```

void main()
{
    struct player p1,p2; FILE *f3;
    f3=fopen("player.txt","w");

    printf("\n Enter details of player name ,age and runs scored :\n ");
    fflush(stdin);
    scanf("%s %d %d",p1.pname,&p1.age,&p1.runs);

    fwrite(&p1,sizeof(p1),1,f3);
    fclose(f3);
    f3=fopen("player.txt","r");
    fread(&p2,sizeof(p2),1,f3);
    fflush(stdout);
    printf("\nPLAYERNAME:=%s\tAGE:=%d\tRUNS:=%d      ",p2.pname,p2.age,p2.runs);
    fclose(f3);
}

```

WRITING AND READING STRUCTURES USING BINARY FILES

RANDOM ACCESS TO FILES :

At times we needed to access only a particular part of a file rather than accessing all the data sequentially, which can be achieved with the help of functions **fseek**, **ftell** and **rewind** available in IO library.

ftell():-

ftell takes a file pointer and returns a number of type **long**, that corresponds to the current position. This function is useful in saving the current position of the file, which can later be used in the program.

Syntax: **n=ftell(fp);**

n would give the Relative offset (In bytes) of the current position. This means that already **n** bytes have a been read or written

rewind():-

It takes a file pointer and resets the position to the start of the file.

Syntax: **rewind(fp);**

n=ftell(fp); would assign 0 to **n** because the file position has been set to start of the file by rewind(). The first byte in the file is numbered 0, second as 1, so on. This function helps in reading the file more than once, without having to close and open the file.

Whenever a file is opened for reading or writing a rewind is done implicitly.

fseek:-

fseek function is used to move the file pointer to a desired location within the file.

Syntax: **fseek(file ptr,offset,position);**

file pointer is a pointer to the file concerned, offset is a number or variable of type long and position is an integer number which takes one of the following values. The offset specifies the number of positions(**Bytes**) to be moved from the location specified by the position which can be positive implies moving forward and negative implies moving backwards.

POSITION VALUE	VALUE CONSTANT	MEANING
0	SEEK_SET	BEGINNING OF FILE
1	SEEK_CUR	CURRENT POSITION
2	SEEK_END	END OF FILE

Example: **fseek(fp,10,0) ;**

fseek(fp,10,SEEK_SET);// file pointer is repositioned in the forward direction 10 bytes.

fseek(fp,-10,SEEK_END); // reads from backward direction from the end of file.

When the operation is successful fseek returns 0 and when we attempt to move a file beyond boundaries fseek returns -1.

Some of the Operations of fseek function are as follows:

STATEMENT	MEANING
fseek(fp,0L,0);	Go to beginning similar to rewind()
fseek(fp,0L,1);	Stay at current position
fseek(fp,0L,2);	Go to the end of file, past the last character of the file.
fseek(fp,m,0);	Move to (m+1) th byte in the file.
fseek(fp,m,1);	Go forward by m bytes
fseek(fp,-m,1);	Go backwards by m bytes from the current position
fseek(fp,-m,2);	Go backwards by m bytes from the end.(positions the file to the m th character from the end)

Program on random access to files:

```
#include<stdio.h>

void main()
{
    FILE *fp;
    char ch;
    fp=fopen("my1.txt","r");
    fseek(fp,21,SEEK_SET);
```

```

while((ch=fgetc(fp))!=EOF)
{
printf(““%c\t”,ch);

printf(““%d\n”,ftell(fp));
}

rewind(fp);

printf(““%d\n”,ftell(fp));

fclose(fp);
}

```

IMPORTANT PROGRAMS 😊

Write a c program to read and display the contents of a file?

```

#include<stdio.h>
void main()
{
FILE *f1;
Char ch;
f1=fopen("data.txt","w");
printf("\n enter some text here and press cntrl D or Z to stop :\n");
while((ch=getchar())!=EOF)
fputc(ch,f1);
fclose(f1);
printf(“\n the contents of file are \n:”);
f1 = fopen(“data.txt”, "r");
while( ( ch = fgetc(f1) ) != EOF )
putchar(ch);
fclose(f1);
}

```

Write a C program to count no.of characters, words , spaces ,lines and occurrence of character in a given file.

```

#include<stdio.h>
void main()
{
FILE *f1;
int char_cnt=0,word_cnt=0,space_cnt=0,line_cnt=0,char_occur=0;
char ch,c;
clrscr();

```

```

printf("\nEnter character to find occurrence:");
c=getchar();
f1=fopen("data.txt","w");
printf("\nEnter some text here and press cntrl D or Z to stop :\n");
while((ch=getchar())!=EOF)
fputc(ch,f1);
fclose(f1);

f1 = fopen("data.txt","r");
while( ( ch = fgetc(f1) ) != EOF )
{
char_cnt++;
if(ch==' ')
word_cnt++;
if(ch==' ')
space_cnt++;
if(ch=='\n')
line_cnt++;
if(ch==c)
char_occur++;

}
fclose(f1);
printf("\nchar_count=%d",char_cnt);
printf("\nword_count=%d",word_cnt*2);
printf("\nspaces_count=%d",space_cnt);
printf("\nlines_count=%d",line_cnt);
printf("\n%c_occurrence=%d",c,char_occur);

getch();
}

```

```

Enter character to find occurrence:d

Enter some text here and press cntrl D or Z to stop :
first line
second line
third line
fourth line→

char_count=46
word_count=8
spaces_count=4
lines_count=4
d_occurrence=2

```

Write a c program to copy the contents of one file to another?

```
#include<stdio.h>

void main()
{
    FILE *f1,*f2;
    char ch;
    f1=fopen("mynew2.txt","w");

    printf("\n enter some text here and press cntrl D or Z to stop :\n");
    while((ch=getchar())!=EOF)
        fputc(ch,f1);

    fclose(f1);
    f1=fopen("mynew2.txt","r");
    f2=fopen("dupmynew2.txt","w");
    while((ch=getc(f1))!=EOF)
        putc(ch,f2);
    fcloseall();

    printf("\n the copied file contents are :");

    f2 = fopen("dupmynew2.txt","r");

    while( ( ch = fgetc(f2) ) != EOF )
        putchar(ch);
    fclose(f2);

}
```

Write a C program to copy data from one file to other, while doing so convert uppercase characters to lowercase OR vice versa.

```
#include<stdio.h>

void main()
{
    FILE *f1,*f2;
    char ch;
    f1=fopen("one.txt","w");
    printf("\n enter some text here and press cntrl D or Z to stop :\n");
    while((ch=getchar())!=EOF)
        fputc(ch,f1);
    fclose(f1);

    f1=fopen("one.txt","r");
    f2=fopen("two.txt","w");
```



```

while((ch=getc(f1))!=EOF)
{
ch=toupper(ch); //for lowercase ch=tolower(ch);
putc(ch,f2);
}
fcloseall();

printf("\n the copied file contents are :");
f2 = fopen("dupmynew2.txt","r");
while( ( ch = fgetc(f2) ) != EOF )
putchar(ch);
fclose(f2);
}

```

Write a c program to merge two files into a third file? (Or)

Write a c program for the following .there are two input files named “first.dat” and “second.dat” .The files are to be merged. That is,copy the contents of first.dat and then second.dat to a new file named result.dat?

```

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

int main()
{
FILE *fs1, *fs2, *fs3;
char ch, file1[20], file2[20], file3[20];
printf("Enter name of first file\n");
gets(file1);
printf("Enter name of second file\n");
gets(file2);
printf("Enter name of file which will store contents of two files\n");
gets(file3);
fs1=fopen(file1,"w");
printf("\n enter some text into file1 here and press cntrl D or Z to stop :\n");
while((ch=getchar())!=EOF)
fputc(ch,fs1); fclose(fs1);

fs2=fopen(file2,"w");
printf("\n enter some text into file2 here and press cntrl D or Z to stop :\n");
while((ch=getchar())!=EOF)
fputc(ch,fs2);
fclose(fs2);

fs1 = fopen(file1,"r");
fs2 = fopen(file2,"r");

```

```

if( fs1 == NULL || fs2 ==
NULL )
{
perror("Error ");
exit(1);
}
fs3 = fopen(file3,"w");
while( ( ch = fgetc(fs1) ) != EOF )
fputc(ch,fs3);
while( ( ch = fgetc(fs2) ) != EOF )
fputc(ch,fs3);

printf("Two files were merged into %s file successfully.\n",file3);
fcloseall();
fs3 = fopen(file3,"r");
printf("\n the merged file contents are :");
while( ( ch = fgetc(fs3) ) != EOF )
putchar(ch);
fclose(ft);
return 0;
}

```

Write a c program to append the contents of a file.

```

#include<stdio.h>
void main()
{
FILE *fp1;
char ch;
fp1=fopen("sunday.txt","w");

printf("\n Enter some Text into file :\n");
while((ch=getchar())!='.')
fputc(ch,fp1);

fclose(fp1);

fp1=fopen("sunday.txt","a+"); //to append
printf("\n Enter some MORE Text into file :\n");
while((ch=getchar())!='.')
fputc(ch,fp1);
rewind(fp1);

printf("\n The complete Text in file is :\n");
while((ch=fgetc(fp1))!=EOF)
putchar(ch);
fclose(fp1);
}

```

Write a C program to copy odd numbers in odd file and even numbers in even file.

Ans: please refer the getw () and putw () topic.

Write a C program to copy structure data into a file.

Ans: please refer the fread () and fwrite () topic.

Write a C program to display the reverse the contents of a file?

```
#include<stdio.h>
#include<stdlib.h>
int main()
{
    FILE *fp1;

    char ch;

    int x;
    fp1=fopen ("any1.txt","w");

    printf("\n Enter some text into file :\n");
    while((ch=getchar())!=EOF)

        fputc(ch,fp1);

    fclose(fp1);
    fp1=fopen("any1.txt","r");
    if(fp1==NULL)
    {
        printf("\n cannot open file:");
        exit(1);
    }
    fseek(fp1,-1L,2);
    x=ftell(fp1);
    printf("\n Th text in the file in reverse order is : \n");
    while(x>=0)
    {

        ch=fgetc(fp1);
        printf("%c",ch);
        x--;
        fseek(fp1,x,0);
    }
    fclose(fp1);
}
```

SHORT ANSWERS ☺

1. Why is it necessary to give size of an array in array declaration? [May 2019]
2. Define the terms: Binary file and text file [may 2019]
3. What is the purpose of feof() function ? [May 2019]
4. How can we determine whether a file is opened successfully opened or not using fopen() function ? [June 2019]
5. Write a C program to read a binary file and print it in console. [June 2019]
6. Write about ftell() function. [Aug 2019]
7. Explain fseek() function with example. [Dec 2018]
8. How to handle errors with file functions? Explain [Aug 2019]
9. Name few preprocessor directives. [Dec 2018]
10. Write the concept of a file.
11. Discuss about the different modes available for opening a file.
12. What is meant by opening a data file? How is this accomplished?

LONG ANSWERS ☺

PREPROCESSOR

1. Explain about preprocessor commands [June 2019] (Or)
List and define the pre-processor statements in C. [Dec 2018] (Or)
Briefly explain the pre-processor directives in detail. 10 M [Aug 2019]

FILES

1. List and explain various file read and write functions available in c with examples illustrating their usage and implementation. 10 M [May 2019] (Or) Explain various standard library functions for handling files.
2. Write the syntax of fseek() function in C and explain the same.[may 2019]
3. How to use fseek() for randomly access the file content? Explain. [Aug 2019]
4. Explain the concept of streams and their significance in I/O operations. [May 2019]
5. Explain steps for file operations and different modes of files. [June 2019]
6. What is a file pointer? Explain steps for sequential file operations. [Dec 2018]
7. Compare gets() and fgets() with and example.[Dec 2018]
8. Write a program to copy the contents of one file to another file. [Dec 2018]
(Or) Write a C program that copies the contents of one file to another file. [Aug 2019]
9. Write a C program to copy the content from one file to another file using fseek() function.
10. Write a program to copy contents of one file to another using file names passed as the command line arguments.
11. Write a program to read a text file, convert all the lowercase characters into uppercase characters in the file. [June 2019]
12. Write a 'C' program to count the number of characters in a file.
13. Write a complete C program for finding the number of words in the given text file. Assume that the words are separated by blanks or tabs.
14. Write a program to find the number of occurrences of a given word in a given file.
15. Write C-language program that reads a C-program file and outputs number of lines in the program.
16. Write a 'C' program to create a file contains a series of integer numbers and then reads all numbers of this file and write all odd numbers to other file called odd and write all even numbers to a file called even.
17. Write a complete C program for the following: There are two input files named "first.dat" and "second.dat". The files are to be merged. That is, copy the content of "first.dat" and then the content of "second.dat" to a new file named "result.dat".
18. Write a C program to replace every 5th character of the data file, using fseek() command
19. Write a C program to copy the binary file from another file.
20. Write a C program to store the structure data into a file.
21. Write a C program read and write the content of the file using fprintf() and fscanf() functions.
22. Differentiate between fprintf and fwrite statements. When do you prefer to use fwrite instead of fprintf ?
23. Briefly explain about positioning functions in files (Or) Explain the following operations (a) fseek()
(b) ftell (c) rewind() (d) ferror()

UNIT-IV

FUNCTIONS AND DYNAMIC MEMORY ALLOCATION

Function in C:

A function is a block of code that performs a specific task. It has a name and it is reusable. It can be executed from as many different parts in a program as required; it can also return a value to calling program.

All executable code resides within a **function**. It takes input, does something with it, then give the answer. A C program consists of one or more functions.

A computer program cannot handle all the tasks by itself. It requests other program like entities called functions in C. We pass information to the function called **arguments** which specified when the function is called. A function either can return a value or returns nothing. Function is a subprogram that helps reduce coding.

Simple Example of Function in C

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

int addition (int, int); //Function Declaration

void main() //Calling Function
{
    int sum;

    sum= addtion(30,3); //Function Call

    printf ("The Result is %d", sum);
}

int addition (int x, int y) //Called Function
{
    int z;
    z=x + y;
    return (z);
} //Function Definition (Or) Function Body
```

Output: The Result is 33

Why to use function:

Basically there are **two reasons** because of which we use functions

1. Writing functions avoids rewriting the same code over and over. For example - if you have a section of code in a program which calculates the area of triangle. Again you want to calculate the area of different triangle then you would not want to write the same code again and again for triangle then you would prefer to jump a "section of code" which calculate the area of the triangle and then jump back to the place where you left off. That section of code is called "function".

2. Using function it becomes easier to write a program and keep track of what they are doing. If the operation of a program can be divided into separate activities, and each activity placed in a different function, then each could be written and checked more or less independently. Separating the code into modular functions also makes the program easier to design, debug and understand.

Types of Function in C:

(i). Library Functions in C

C provides library functions for performing some operations. These functions are present in the c library and they are predefined.

For example `sqrt()` is a mathematical library function which is used for finding the square root of any number. The function `scanf` and `printf()` are input and output library function similarly we have `strcmp()` and `strlen()` for string manipulations. To use a library function we have to include some header file using the preprocessor directive `#include`.

For example to use input and output function like `printf()` and `scanf()` we have to include `stdio.h`, for math library function we have to include `math.h` for string library `string.h` should be included.

(ii). User Defined Functions in C

A user can create their own functions for performing any specific task of program are called user defined functions. To create and use these function we have to know these 3 elements.

1. Function Declaration
2. Function Definition
3. Function Call

1. Function declaration

The program or a function that calls a function is referred to as the calling program or calling function. The calling program should declare any function that is to be used later in the program this is known as the function declaration or function prototype.

Syntax:

```
return_type  function_name (argumentlist);
```

This declaration should be above the calling function.

2. Function Definition

The function definition consists of the whole description and code of a function. It tells that what the function is doing and what are the input outputs for that. A function is called by simply writing the name of the function followed by the argument list inside the parenthesis.

Syntax:

```
return_type  function_name (argumentlist)  
{  
    -----  
    -----  
    -----  
}
```

Example:

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

Function definitions have two parts:

Function Header

The first line of code is called Function Header.

```
int sum( int x, int y)
```

It has three parts

- (i). The name of the function i.e. sum
- (ii). The parameters of the function enclosed in parenthesis
- (iii). Return value type i.e. int

Function Body

Whatever is written within { } is the body of the function.

3. Function Call

In order to use the function we need to invoke it at a required place in the program. This is known as the function call.

calling function: a function which calls/invokes the other function is called **calling function**.

called function: a function which is being invoked/called **called function**

actual parameters:

The list of variables in calling function is known as **actual parameters**.

Actual parameters are variables that are declared in function call.

Actual parameters are passed without using type.

```
main( )
```

```
{ .....
```

```
function_name (actual parameters);
```

```
.....
```

```
}
```

formal parameters:

The list of variables in called function is known as **formal parameters**.

Formal parameters are variables that are declared in the header of the function definition.

Formal parameters have type preceding with them.

```
return_type function_name (formal parameters)
```

```
{ .....
```

```
Function body;
```

```
.....
```

```
}
```

Properties of Functions

- Every function has a unique name. This name is used to call function from “main()” function.
- A function performs a specific task.
- A function returns a value to the calling program.

Advantages of Functions in C

- Functions has top down programming model. In this style of programming, the high level logic of the overall problem is solved first while the details of each lower level functions is solved later.
- A C programmer can use function written by others
- Debugging is easier in function
- It is easier to understand the logic involved in the program
- Testing is easier

THE VARIOUS CATEGORIES OF USER DEFINED FUNCTIONS IN C WITH EXAMPLE:

A function depending on whether arguments are present or not and whether a value is returned or not may belong to any one of the following categories:

- (i) Functions with no arguments and no return values.
- (ii) Functions with arguments and no return values.
- (iii) Functions with arguments and return values.
- (iv) Functions with no arguments and return values.

(i) Functions with no arguments and no return values:-

When a function has no arguments, it does not return any data from calling function. When a function does not return a value, the calling function does not receive any data from the called function. That is there is no data transfer between the calling function and the called function.

Example

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

void add();

void main()
{
    add();
}

void add()
{
    int x=10,y=20,z;
    z=x+y;
    printf ("sum=%d",z);
}
```

Output : sum=30.

(ii) Functions with arguments and no return values:-

When a function has arguments data is transferred from calling function to called function. The called function receives data from calling function and does not send back any values to calling function. Because it doesn't have return value.

Example

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

void add(int , int);

void main()
{
int a=10,b=20;
add(a,b);
}

void add(int x,int y)
{
int z;
z=x+y;
printf ("sum=%d",z);
}
```

Output : sum=30.

(iii) Function with no arguments and return type:-

When function has no arguments data cannot be transferred to called function. But the called function can send some return value to the calling function.

Example

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

int add();

void main()
{
int sum;
sum=add();
printf ("sum=%d",sum);
}
```

```

int add()
{
int x=10,y=20,z;
z=x+y;

return z;
}

```

Output : sum=30.

(iv) Functions with arguments and return values:-

In this data is transferred between calling and called function. That means called function receives data from calling function and called function also sends the return value to the calling function.

Example

```

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

int add(int , int);

void main()
{
int a=10,b=20,sum;

sum=add(a,b);

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

```

```

int add(int x , int y)
{
int z;
z=x+y;
return z;
}

```

Output : sum=30.

PARAMETER PASSING MECHANISMS IN C-LANGUAGE WITH EXAMPLES ☺

Most programming languages have 2 strategies to pass parameters. They are

- (i) pass by value / call by value / copy by value:
- (ii) pass by reference / call by reference / copy by address:

(i) Pass by value (or) call by value (or) copy by value:-

In this method calling function sends a copy of actual values to called function, but the changes in called function formal parameters does not reflect the actual (original) parameters of calling function.

Example program:

```
#include<stdio.h>
void swap(int , int);
void main( )
{
int a=10, b=20;
swap(a,b);
printf("a=%d,b=%d", a,b);
}

void swap(int x, int y)
{
int temp;
temp=x
x=y;
y= temp;
printf("x=%d,y=%d", x,y);
}
```

Output: a=10 b=20

x=20 y=10

The result clearly shown that the called function does not reflect the original values in main function.

(ii) Pass by reference (or) call by address (or) copy by address:-

In this method calling function sends address of actual values as a parameter to called function, called function performs its task and sends the result back to calling function. Thus, the changes in called function formal parameters reflect the actual (original) parameters of calling function. To return multiple values from called to calling function we use pointer variables.

Calling function needs to pass ‘&’ operator along with actual arguments and called function need

to use '*' operator along with formal arguments. Changing data through an address variable is known as indirect access and '*' is represented as indirection operator.

Example program:

```
#include<stdio.h>
void swap(int* , int*);
void main( )
{
    int a=10, b=20;
    swap(&a,&b);
    printf("a=%d,b=%d", a,b);
}

void swap(int *x, int *y)
{int t;
  t=*x
  *x=*y;
  *y= t;
  printf("x=%d,y=%d", *x,*y);
}
```

Output: a=20 b=10

x=20 y=10

The result clearly shown that the called function reflects the original values in main function. So that it changes original values.

Differentiate actual parameters and formal parameters.

Actual parameters	Formal parameters
The list of variables in calling function is known as actual parameters .	The list of variables in called function is known as formal parameters .
Actual parameters are variables that are declared in function call.	Formal parameters are variables that are declared in the header of the function definition.
Actual parameters are passed without using type	Formal parameters have type preceeding with them.
main() { function_name (actual parameters); }	return_type function_name(formal parameters) { function body; }
NOTE: Formal and actual parameters must match exactly in type, order, and number. Formal and actual parameters need not match for their names.	

Passing Arrays to functions:

To process arrays in a large program, we need to pass them to functions. We can pass arrays in two ways:

- 1) pass individual elements
- 2) pass the whole array.

Passing Individual Elements:

One-dimensional Arrays:

We can pass individual elements by either passing their data values or by passing their addresses. We pass data values i.e; individual array elements just like we pass any data value .As long as the array element type matches the function parameter type, it can be passed. The called function cannot tell whether the value it receives comes from an array, a variable or an expression.

Ex: Program using call by value

```
void fun(int x);

void main()
{
    int a[5]={ 1,2,3,4,5};
    fun(a[3]);
}

void fun( int x)
{
    printf("%d",x);
}
```

Output: 4

Two-dimensional Arrays:

The individual elements of a 2-D array can be passed in the same way as the 1-D array. We can pass 2-D array elements either by value or by address.

Ex: Program using call by value

```

void fun(int x);
main()
{
int a[2][2]={1,2,3,4};
fun (a[0][1]);
}

```

```

void fun(int x)
{
printf(“%d”,x);
}

```

Output: 2

Passing an entire array to a function:

One-dimensional array:

To pass the whole array we simply use the array name as the actual parameter. In the called function, we declare that the corresponding formal parameter is an array. We do not need to specify the number of elements.

Program :

```

void fun(int x[]) ;

void main()
{
int a[5]={ 1,2,3,4,5};
fun(a);
}

void fun( int x[])
{
int i,sum=0;
for(i=0;i<5;i++)
sum=sum+a[i];
printf(“%d”,sum);
}

```

Output: 15

TWO-dimensional array:

When we pass a 2-D array to a function, we use the array name as the actual parameter just as we did with 1-D arrays. The formal parameter in the called function header, however must indicate that the array has two dimensions.

Rules:

- The function must be called by passing only the array name.
- In the function definition, the formal parameter is a 2-D array with the size of the second dimension specified.

Program:

```
void fun(int a[][2]) ;
void main()
{
    int a[2][2]={ 1,2,3,4};
    fun(a);
}
```

```
void fun(int x[][2])
{
    int i,j;
    for(i=0;i<2;i++)
    for(j=0;j<2;j++)
    printf(“ %d”,x[i][j]);
}
```

Output: 1 2 3 4

PASSING POINTERS TO FUNCTIONS: ☺

Please refer Unit-iv **Pass by reference (or) call by address (or) copy by address** topic.

STANDARD FUNCTIONS AND LIBRARIES:

C functions can be classified into two categories, namely, library functions and user-defined functions. `main` is the example of user-defined functions. `printf` and `scanf` belong to the category of library functions. The main difference between these two categories is that library functions are not required to be written by us whereas a user-defined function has to be developed by the user at the time of writing a program. However, a user-defined function can later become a part of the C program library.

ANSI C Standard Library Functions:

The C language is accompanied by a number of library functions that perform various tasks. The ANSI committee has standardized header files which contain these functions.

Some of the Header files are:

<math.h>	Mathematical functions
<stdio.h>	Standard I/O library functions
<stdlib.h>	Utility functions such as string conversion routines memory allocation routines , random number generator , etc.
<string.h>	String Manipulation functions
<time.h>	Time Manipulation functions

MATH.H

The math library contains functions for performing common mathematical operations.

abs :	Returns the absolute value of an integer x
cos :	Returns the cosine of x, where x is in radians
exp :	Returns "e" raised to a given power
fabs :	Returns the absolute value of a float x
log :	Returns the logarithm to base e
log10 :	Returns the logarithm to base 10
pow :	Returns a given number raised to another number
sin :	Returns the sine of x, where x is in radians
sqrt :	Returns the square root of x
tan :	Returns the tangent of x, where x is in radians
ceil :	The ceiling function rounds a number with a decimal part up to the next highest integer (written mathematically as $\lceil x \rceil$)
floor :	The floor function rounds a number with a decimal part down to the next lowest integer (written mathematically as $\lfloor x \rfloor$)

STDIO.H

Standard I/O library functions

printf:	Formatted printing to stdout
scanf:	Formatted input from stdin
fprintf:	Formatted printing to a file
fscanf:	Formatted input from a file
getc:	Get a character from a stream (e.g., stdin or a file)
putc:	Write a character to a stream (e.g., stdout or a file)
fgets:	Get a string from a stream
fputs:	Write a string to a stream
fopen:	Open a file
fclose:	Close a file

STDLIB.H

Utility functions such as string conversion routines, memory allocation routines, random number generator, etc.

atof:	Convert a string to a double (not a float)
atoi:	Convert a string to an int
exit:	Terminate a program, return an integer value
free:	Release allocated memory
malloc:	Allocate memory
rand:	Generate a pseudo-random number
system:	Execute an external command

STRING.H

There are many functions for manipulating strings. Some of the more useful are:

strcat :	Concatenates (i.e., adds) two strings
strcmp:	Compares two strings
strcpy:	Copies a string
strlen:	Calculates the length of a string (not including the null)
strstr:	Finds a string within another string
strtok:	Divides a string into tokens (i.e., parts)

TIME.H

This library contains several functions for getting the current date and time.

time:	Get the system time
ctime:	Convert the result from time() to something meaningful

RECURSION: ☺

Recursion is a process by which a function calls itself repeatedly, until some specified condition has been satisfied.

When a function calls itself, a new set of local variables and parameters are allocated storage on the stack, and the function code is executed from the top with these new variables. A recursive call does not make a new copy of the function. Only the values being operated upon are new. As each recursive call returns, the old local variables and parameters are removed from the stack, and execution resumes immediately after the recursive call inside the function.

The main advantage of recursive functions is that we can use them to create clearer and simpler versions of several programs.

Syntax:-

A function is **recursive** if it can call itself; either directly:

```
void f( )  
{  
f();  
}
```

(or) indirectly:

```
void f( )  
{  
g();  
}
```

```
void g( )  
{  
f();  
}
```

Recursion rule 1: Every recursive method must have a **base case** -- a condition under which no recursive call is made -- to prevent infinite recursion.

Recursion rule 2: Every recursive method must make progress toward the base case to prevent infinite recursion

Write a program to find factorial of a number using recursion.

```
#include<stdio.h>
int fact(int);
main()
{
int n,f;
printf("\n Enter any number:");
scanf("%d",&n);
f=fact(n);
printf("\n Factorial of %d is %d",n,f);
}

int fact(int n)
{
int f;
if(n==0||n==1) //base case
f=1;
else
f=n*fact(n-1); //recursive case
return f;
}
```

Output:-

Enter any number: 5
Factorial of 5 is 120

Write a program to generate Fibonacci series using recursive functions.

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

int fib(int x);
void main()
{
int n,i,fib;
printf("Enter the limit:");
scanf("%d",&n);
printf("The fibonacci series is:");
for(i=0;i<n;i++)
{
```

```

fib=fib(i);
printf("%d\t",fib);
}
}

int fib(int x)
{
if(x==0||x==1)
return x;
else
return fib(x-1)+fib(x-2);
}

```

Output:-

Enter the limit: 5

The fibonacci series is: 0 1 1 2 3

Differentiate between recursion and non- recursion:

Recursion: - Recursion is a process by which a function calls itself repeatedly, until some specified condition has been satisfied.

When a function calls itself, a new set of local variables and parameters are allocated storage on the stack, and the function code is executed from the top with these new variables. A recursive call does not make a new copy of the function. Only the values being operated upon are new. As each recursive call returns, the old local variables and parameters are removed from the stack, and execution resumes immediately after the recursive call inside the function.

Ex:-

```

void main( )
{
int n=5;
fact( n);
}

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

```

Non-Recursion:-

Using looping statements we can handle repeated statements in „C“.

The example of non recursion is given below.

Ex:-

```
void main( )
{
int n=5,res;
res = fact(n);
printf(“%d”,res);
}
```

```
int fact( )
{
int f=1;
for(i=1;i<=n;i++)
f=f*i;
return f;
}
```

Differences:

- Recursive version of a program is slower than iterative version of a program due to overhead of maintaining stack.
- Recursive version of a program uses more memory (for the stack) than iterative version of a program.
- Sometimes, recursive version of a program is simpler to understand than iterative version of a program.

Limitations of recursive functions:

- Recursive solution is always logical and it is very difficult to trace.(debug and understand).
- In recursive we must have an if statement somewhere to force the function to return without the recursive call being executed, otherwise the function will never return.
- Recursion takes a lot of stack space, usually not considerable when the program is small and running on a PC.
- Recursion uses more processor time.

SOME IMPORTANT PROGRAMS ON RECURSION

Write a program to calculate GCD of two numbers using recursion.

```
#include <stdio.h>
#include <conio.h>
int gcd(int n1, int n2);

void main()
{
    int n1,n2;
    clrscr();
    printf("Enter two positive integers: ");
    scanf("%d %d", &n1, &n2);
    printf("G.C.D of %d and %d is %d.", n1, n2, gcd(n1,n2));
    getch();
}

int gcd(int num1, int num2)
{
    if(num2 == 0)
        return num1;
    else
        return gcd(num2, num1%num2);
}
```

Output:

Enter two positive integers: 8 12
G.C.D of 8 and 12 is 4.

Write C program to reverse the given number using recursion

```
#include<stdio.h>
void main(){
    int num,reverse_number;
    clrscr();
    //User would input the number
    printf("\nEnter any number:");
    scanf("%d",&num);

    //Calling user defined function to perform reverse
    reverse_number=reverse(num);
    printf("\nAfter reverse the no is :%d",reverse_number);
    getch();
}
```



```

int sum=0,rem;
reverse(int num)
{
    if(num)
    {
        rem=num%10;
        sum=sum*10+rem;
        reverse(num/10);
    }
    else
        return sum;
}

```

Output:

Enter any number:123
 After reverse the no is :321

Write a program for Sum of Natural Numbers Using Recursion

```

#include <stdio.h>
int addNumbers(int n);

void main()
{
    int num;
    clrscr();
    printf("Enter a positive integer: ");
    scanf("%d", &num);
    printf("Sum = %d",addNumbers(num));
    getch();
}

int addNumbers(int n)
{
    if(n <=1)
        return n;
    else
        return n+addNumbers(n-1);
}

```

Output:

Enter a positive integer: 5
 Sum = 15

Write a recursive function for swapping of two numbers (Practice)

Program for Tower of Hanoi ☺

Tower of Hanoi is a mathematical puzzle where we have three rods and n disks. The objective of the puzzle is to move the entire stack to another rod, obeying the following simple rules:

- 1) Only one disk can be moved at a time.
- 2) Each move consists of taking the upper disk from one of the stacks and placing it on top of another stack i.e. a disk can only be moved if it is the uppermost disk on a stack.
- 3) No disk may be placed on top of a smaller disk.
- 4) All disks should be moved from source to destination with few moves.

Take an example for 1 disks :

Let rod 1 = 'A', rod 2 = 'B', rod 3 = 'C'.

Step 1 : Move 1 disk from A to C

Take an example for 2 disks :

Let rod 1 = 'A', rod 2 = 'B', rod 3 = 'C'.

Step 1 : Move 1 disk from A to B.

Step 2 : Move 2 disk from A to C.

Step 3 : Move 1 disk from B to C.

Take an example for 3 disks :

Let rod 1 = 'A', rod 2 = 'B', rod 3 = 'C'.

Step 1 : Move disk 1 from rod A to rod C

Step 2 : Move disk 2 from rod A to rod B

Step 3 : Move disk 1 from rod C to rod B

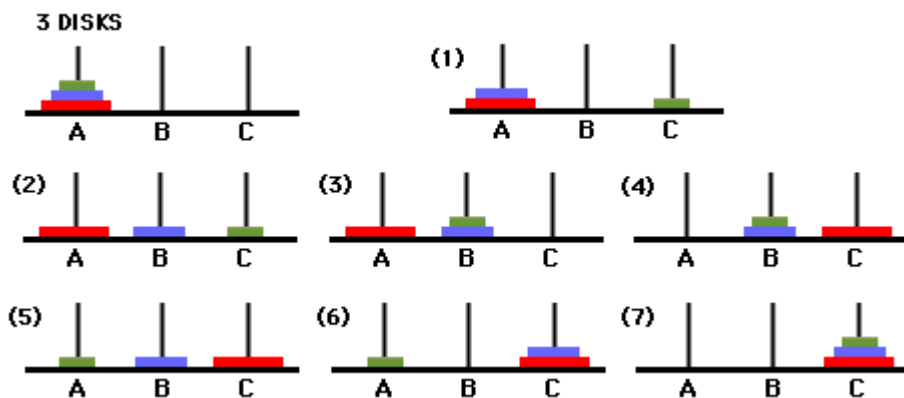
Step 4 : Move disk 3 from rod A to rod C

Step 5 : Move disk 1 from rod B to rod A

Step 6 : Move disk 2 from rod B to rod C

Step 7 : Move disk 1 from rod A to rod C

Image illustration for 3 disks :



The pattern here is :

Let A =Source(s) B=Auxiliary (A) C=Destination (D) .

Step 1: n=1 then Move disk from S to D

Step 2: n>1 then Move (n-1) disks from S to A using D

Step 3: Move disk from S to D

Step 4: Move (n-1) disks from A to D using S

Write a C program for Towers of Hanoi.

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

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

```
void TOH(int n, char S,char D, char Aux)
```

```
{
    if (n == 1)
    {
        printf("\nMove disk %d from rod %c to rod %c.",n,S,D);
        return;
    }
    TOH(n-1, S, Aux, D);
    printf("\nMove disk %d from rod %c to rod %c",n,S,D);
    TOH(n-1, Aux, D, S);
}
```

```
// Driver code
```

```
void main()
{
    int n; // Number of disks
    clrscr();
    printf("\nEnter number of disks: ");
    scanf("%d",&n);
    TOH(n, 'A', 'C', 'B'); // A, B and C are names of rods
    getch();
}
```

Output 1:

Enter number of disks: 2

Move disk 1 from rod A to rod B
Move disk 2 from rod A to rod C
Move disk 1 from rod B to rod C

Output 2:

Enter number of disks: 3

Move disk 1 from rod A to rod C
Move disk 2 from rod A to rod B
Move disk 1 from rod C to rod B
Move disk 3 from rod A to rod C
Move disk 1 from rod B to rod A
Move disk 2 from rod B to rod C
Move disk 1 from rod A to rod C

DYNAMIC MEMORY ALLOCATION

The exact size of array is unknown until the compile time, i.e., time when a compiler compiles code written in a programming language into an executable form. The size of array you have declared initially can be sometimes insufficient and sometimes more than required. Dynamic memory allocation allows a program to obtain more memory space, while running or to release space when no space is required.

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

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

[malloc\(\)](#)

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

Syntax of malloc()

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

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

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

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

calloc()

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

Syntax of calloc()

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

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

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

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

free()

Dynamically allocated memory with either calloc() or malloc() does not get return on its own. The programmer must use free() explicitly to release space.

syntax of free()

free(ptr);

This statement causes the space in memory pointer by ptr to be deallocated.

Examples of calloc() and malloc()

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

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

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

```
int main(){
```

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

```
    printf("Enter number of elements: ");
```

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

```
    ptr=(int*)malloc(n*sizeof(int)); //memory allocated using malloc
```

```
    if(ptr==NULL)
```

```
    {
```

```
        printf("Error! memory not allocated.");
```

```
        exit(0);
```

```
    }
```

```
    printf("Enter elements of array: ");
```

```

for(i=0;i<n;++i)
{
    scanf("%d",ptr+i);
    sum+=*(ptr+i);
}

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

```

Output:

Enter number of elements: 5

Enter elements of array: 1 2 3 4 5

Sum=15

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

```

#include<stdio.h>
#include <stdlib.h>
int main(){
    int n,i,*ptr,sum=0;

    printf("Enter number of elements: ");
    scanf("%d",&n);
    ptr=(int*)calloc(n,sizeof(int));
    if(ptr==NULL)
    {
        printf("Error! memory not allocated.");
        exit(0);
    }

    printf("Enter elements of array: ");
    for(i=0;i<n;++i)
    {

        scanf("%d",ptr+i);
        sum+=*(ptr+i);
    }

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

```

Output:

Enter number of elements: 5
Enter elements of array: 1 2 3 4 5
Sum=15

realloc():

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

Syntax of realloc()

ptr=realloc(ptr,newsize);

Here, *ptr* is reallocated with size of newsize.

```
#include <stdio.h>
#include <stdlib.h>
int main(){
    int *ptr,i,n1,n2;
    clrscr();
    printf("Enter size of array: ");
    scanf("%d",&n1);
    ptr=(int*)malloc(n1*sizeof(int));
    printf("Address of previously allocated memory: ");
    for(i=0;i<n1;++i)
        printf("%u\t",ptr+i);
    printf("\nEnter new size of array: ");
    scanf("%d",&n2);
    ptr=realloc(ptr,n2);

    printf("Address of newly allocated memory: ");
    for(i=0;i<n1+n2;++i)
        printf("%u\t",ptr+i);
    getch();
    return 0;
}
```

```
Enter size of array: 5
Address of previously allocated memory: 1964    1966    1968    1970    1972

Enter new size of array: 3
Address of newly allocated memory: 1964 1966    1968    1970    1972    1974
1976    1978    _
```

SHORT ANSWERS ☺

1. Explain about parameter passing using copy by address.[Dec 2018]
2. Give syntax to pointer to function. [June 2019]
3. How does a recursive function is differ from an iterative function? [May 2019]
4. List the limitations of recursive function.[Aug 2019]
5. Write recursive function for Towers of Hanoi.[Dec 2018]
6. Write a short note on dynamic memory allocation.[June 2019]
7. Write the syntax and purpose of malloc() function.[May 2019]
8. Discuss about allocating and freeing memory.[Aug 2019]
9. List the advantages of functions.

LONG ANSWERS ☺

FUNCTIONS

1. What is a function prototype? Give an example. [Dec 2018]
2. State the need of user defined functions. [May 2019]
3. Explain the call by value and call by reference parameter passing methods. [May 2019]
(Or) Explain about different parameter passing mechanisms with examples. [Dec 2018]
(Or) What is the difference between actual and formal parameters? With illustrative examples explain parameter passing techniques.
4. How to pass the structure to functions as an argument? Explain with a suitable example. [June 2019]
(Or) How can an entire structure be passed to a function?
5. Explain with examples how arrays are passed as arguments in functions. (10 M) [Aug 2019]
6. List and explain the some C standard functions and libraries. (10 M) [Aug 2019]
7. What are the different standard library functions available in 'C'? Explain with a sample program.
8. What is a function? What are the different types of functions? Explain function with no argument and no return type with an example.

RECURSION

1. Write a C program to generate Fibonacci series using recursive function. [May 2019] [Dec 2018]
2. What is recursion? Write a C program for Towers of Hanoi. Also specify in diagram for it.(10 M) [June 2019]
3. Define Recursion. Write a recursive function for swapping of two numbers. [Dec 2018]
4. Write a C program to find factorial of a given number using recursive function.
5. What do you mean by recursion? What conditions should be mandatory for writing a recursive function? Explain using a suitable C program.
6. Using recursive function for factorial, explain the execution of the function call factorial(5) using stack.

DYNAMIC MEMORY ALLOCATION

1. List and explain the functions used to allocate and free memory dynamically.[May 2019]
2. Explain about allocating memory for arrays of different data types.[June 2019]
3. What is the difference between calloc and malloc functions?
4. Explain about memory allocation functions in C.

UNIT-V

Write an algorithm to find all roots of a quadratic equation $ax^2+bx+c=0$.

Step 1: Start
Step 2: Declare variables a, b, c, D, x1, x2, rp and ip;
Step 3: Calculate discriminant $D \leftarrow b^2 - 4ac$
Step 4: If $D \geq 0$
 $r1 \leftarrow (-b + \sqrt{D}) / 2a$
 $r2 \leftarrow (-b - \sqrt{D}) / 2a$
 Display r1 and r2 as roots.
Else
 Calculate real part and imaginary part
 $rp \leftarrow -b / 2a$
 $ip \leftarrow \sqrt{-D} / 2a$
 Display $rp + j(ip)$ and $rp - j(ip)$ as roots
Step 5: Stop

Write an algorithm to find the largest among three different numbers entered by user.

Step 1: Start
Step 2: Declare variables a, b and c.
Step 3: Read variables a, b and c.
Step 4: If $a > b$
 If $a > c$
 Display a is the largest number.
 Else
 Display c is the largest number.
Else
 If $b > c$
 Display b is the largest number.
 Else
 Display c is the greatest number.
Step 5: Stop

Write an algorithm to check whether a number entered by user is prime or not.

Step 1: Start
Step 2: Declare variables n, i, cnt.
Step 3: Initialize variables $flag \leftarrow 1$ $i \leftarrow 1$ $cnt \leftarrow 0$
Step 4: Read n from user.
Step 5: Repeat the steps until $i \leq n$
 5.1 If remainder of $n \div i$ equals 0
 $cnt \leftarrow cnt + 1$
 5.2 $i \leftarrow i + 1$
Step 6: If $cnt = 2$
 Display n is prime
Else
 Display n is not a prime
Step 7: Stop

BASIC CONCEPT OF ORDER OF COMPLEXITY THROUGH EXAMPLE PROGRAMS

Algorithmic complexity is concerned about how fast or slow particular algorithm performs. We define complexity as a numerical function $T(n)$ - time versus the input size n . We want to define time taken by an algorithm without depending on the implementation details. But you agree that $T(n)$ does depend on the implementation! A given algorithm will take different amounts of time on the same inputs depending on such factors as: processor speed; instruction set, disk speed, brand of compiler and etc. The way around is to estimate efficiency of each algorithm *asymptotically*. We will measure time $T(n)$ as the number of elementary "steps" (defined in any way), provided each such step takes constant time.

Let us consider two classical examples: addition of two integers. We will add two integers digit by digit (or bit by bit), and this will define a "step" in our computational model. Therefore, we say that addition of two n -bit integers takes n steps. Consequently, the total computational time is $T(n) = c * n$, where c is time taken by addition of two bits. On different computers, addition of two bits might take different time, say c_1 and c_2 , thus the addition of two n -bit integers takes $T(n) = c_1 * n$ and $T(n) = c_2 * n$ respectively. This shows that different machines result in different slopes, but time $T(n)$ grows linearly as input size increases.

The process of abstracting away details and determining the rate of resource usage in terms of the input size is one of the fundamental ideas in computer science.

Asymptotic Notations

The goal of computational complexity is to classify algorithms according to their performances. We will represent the time function $T(n)$ using the "big-O" notation to express an algorithm runtime complexity. For example, the following statement

$$T(n) = O(n^2)$$

Says that an algorithm has a quadratic time complexity.

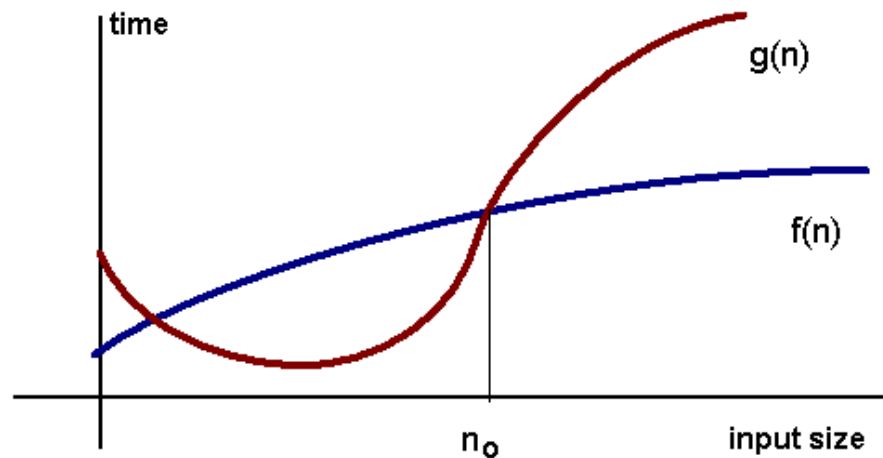
Definition of "big Oh"

For any monotonic functions $f(n)$ and $g(n)$ from the positive integers to the positive integers, we say that $f(n) = O(g(n))$ when there exist constants $c > 0$ and $n_0 > 0$ such that

$$f(n) \leq c * g(n), \text{ for all } n \geq n_0$$

Intuitively, this means that function $f(n)$ does not grow faster than $g(n)$, or that function $g(n)$ is an **upper bound** for $f(n)$, for all sufficiently large $n \rightarrow \infty$

Here is a graphic representation of $f(n) = O(g(n))$ relation:



Examples:

- $1 = O(n)$
- $n = O(n^2)$
- $\log(n) = O(n)$
- $2n + 1 = O(n)$

The "big-O" notation is not symmetric: $n = O(n^2)$ but $n^2 \neq O(n)$.

Exercise. Let us prove $n^2 + 2n + 1 = O(n^2)$. We must find such c and n_0 that $n^2 + 2n + 1 \leq c \cdot n^2$. Let $n_0=1$, then for $n \geq 1$

$$1 + 2n + n^2 \leq n + 2n + n^2 \leq n^2 + 2n^2 + n^2 = 4n^2$$

Therefore, $c = 4$.

Constant Time: $O(1)$

An algorithm is said to run in constant time if it requires the same amount of time regardless of the input size. Examples:

- array: accessing any element
- fixed-size stack: push and pop methods
- fixed-size queue: enqueue and dequeue methods

Linear Time: $O(n)$

An algorithm is said to run in linear time if its time execution is directly proportional to the input size, i.e. time grows linearly as input size increases. Examples:

- array: linear search, traversing, find minimum
- ArrayList: contains method
- queue: contains method

Logarithmic Time: $O(\log n)$

An algorithm is said to run in logarithmic time if its time execution is proportional to the logarithm of the input size. Example:

- binary search

Recall the "twenty questions" game - the task is to guess the value of a hidden number in an interval. Each time you make a guess, you are told whether your guess is too high or too low. Twenty questions game implies a strategy that uses your guess number to halve the interval size. This is an example of the general problem-solving method known as **binary search**:

locate the element a in a sorted (in ascending order) array by first comparing a with the middle element and then (if they are not equal) dividing the array into two subarrays; if a is less than the middle element you repeat the whole procedure in the left subarray, otherwise - in the right subarray. The procedure repeats until a is found or subarray is a zero dimension.

Note, $\log(n) < n$, when $n \rightarrow \infty$. Algorithms that run in $O(\log n)$ does not use the whole input.

Quadratic Time: $O(n^2)$

An algorithm is said to run in logarithmic time if its time execution is proportional to the square of the input size. Examples:

- bubble sort, selection sort, insertion sort

Definition of "big Omega"

We need the notation for the **lower bound**. A capital omega Ω notation is used in this case. We say that $f(n) = \Omega(g(n))$ when there exist constant c that $f(n) \geq c \cdot g(n)$ for all sufficiently large n . Examples

- $n = \Omega(1)$
- $n^2 = \Omega(n)$
- $n^2 = \Omega(n \log(n))$
- $2n + 1 = O(n)$

Definition of "big Theta"

To measure the complexity of a particular algorithm, means to find the upper and lower bounds. A new notation is used in this case. We say that $f(n) = \Theta(g(n))$ if and only $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$. Examples

- $2n = \Theta(n)$
- $n^2 + 2n + 1 = \Theta(n^2)$

Analysis of Algorithms

The term analysis of algorithms is used to describe approaches to the study of the performance of algorithms. In this course we will perform the following types of analysis:

- the worst-case runtime complexity of the algorithm is the function defined by the maximum number of steps taken on any instance of size a .
- the best-case runtime complexity of the algorithm is the function defined by the minimum number of steps taken on any instance of size a .
- the average case runtime complexity of the algorithm is the function defined by an average number of steps taken on any instance of size a .
- the amortized runtime complexity of the algorithm is the function defined by a sequence of operations applied to the input of size a and averaged over time.

Example. Let us consider an algorithm of sequential searching in an array of size n .

Its *worst-case runtime complexity* is $O(n)$

Its *best-case runtime complexity* is $O(1)$

Its *average case runtime complexity* is $O(n/2)=O(n)$

SEARCHING METHODS

Searching is the process of finding a given value position in a list of values. It decides whether a search key is present in the data or not.

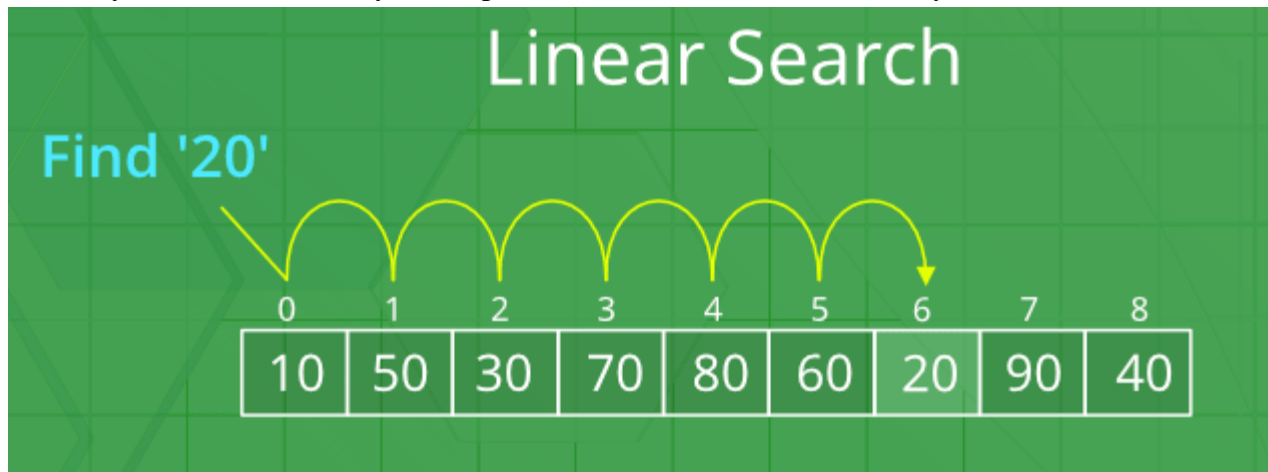
Linear Search

Linear search sequentially checks each element of the list until it finds an element that matches the key value. If the algorithm reaches the end of list, then it returns that the element is not found.

Example:

For example an array consists of 10 elements 10,50,30,70,80,60,20,90,40.

If the key value is 20, then key is compared with each element in the array until it is found.



Algorithm :

Linear_Search(A, n, key)

Step 1: Repeat For $I = 0$ to $N-1$

 Begin

Step 2: do if $\text{key} = A[i]$

Step 3: then display “key is found” and go to step 5

 End For

Step 4: display “key is not found”

Step 5: Exit

Program:

```
void main()
{
    int ar[10], n, i, key;
```

```
clrscr();
printf("Enter the No.Of elements:");
scanf("%d",&n);
printf("Enter %d elements:",n);
for(i=0;i<n;i++)
scanf("%d",&ar[i]);
printf("\nEnter Key:");
scanf("%d",&key);

//linear search logic
for(i=0;i<n;i++)
if(key==ar[i])
{
printf("\nThe key element is found at %d position.",i+1);
getch();
exit(0);
}
printf("\nThe element is not found.");
getch();
}
```

Output:

Enter the No. Of elements:9
Enter 9 elements:10 50 30 70 80 60 20 90 40

Enter Key:20

The key element is found at 7 position.

Complexity of Linear_Search:

The worst case complexity of linear search is $O(n)$.

Binary Search

Binary search is used to find an element in a sorted array. If the array isn't sorted, you must sort it using a sorting technique. If the element to search is present in the list, then we print its location. The program assumes that the input numbers are in ascending order.

1. In this method the given sorted list of **n** elements will be divided into 3 parts
The first element is treated as **low** = 0
The last element is treated as **high** = n-1
And the **mid** = (low + high)/2
2. The key value is compared with the middle value, if the element is found the key index is returned.
3. If it doesn't match then the required element must be either in the left half (or) right half of the middle.
4. If the key is less than the middle value the key is searched in the left part and **high=mid-1**.
Else right part of the middle and **low=mid+1**.
5. With **low** and **high** again above steps are repeated until an element is found.

Example:

For example an array consists of 10 elements

And key is 23 then

Binary Search

	0	1	2	3	4	5	6	7	8	9
Search 23	2	5	8	12	16	23	38	56	72	91
	L=0	1	2	3	M=4	5	6	7	8	H=9
23 > 16 take 2 nd half	2	5	8	12	16	23	38	56	72	91
	0	1	2	3	4	L=5	6	M=7	8	H=9
23 > 56 take 1 st half	2	5	8	12	16	23	38	56	72	91
	0	1	2	3	4	L=5, M=5	H=6	7	8	9
Found 23, Return 5	2	5	8	12	16	23	38	56	72	91

Algorithm :**Binary-Search [A, n,key]**

Step 1: Set low to 0

Step 2: Set high to n-1

Step 3: while low <= high

Step 4; do mid =(low +high)/2

Step 5: if key =A[mid]

Step 6: then display "The key element is found" and go to step 10

Step 7: else if key >A[mid]

Step 8: then low = mid + 1

Step 9: else high = mid - 1

End While

Step 10: Display "Key not found"

Step 11: Exit

Program:

```
//Bubble sort
```

```
void main()
```

```
{
```

```
int ar[10],n,i,j,low,high,mid,key;
```

```
clrscr();
```

```
printf("Enter the No.Of elements:");
```

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

```
printf("Enter %d elements:",n);
```

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

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

```
printf("Enter Key:");
```

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

```
//Bubble sort logic
```

```
low=0;
```

```
high=n-1;
```

```
while(low<=high)
```

```

{
mid=(low+high)/2;
if(key==ar[mid])
{
printf("\nThe key element is found at %d position.",mid+1);
getch();
exit(0);
}
else if(key<ar[mid])
high=mid-1;
else
low=mid+1;
}
printf("\nThe element is not found.");
getch();
}

```

Output:

Enter the No.Of elements:10

Enter %d elements: 2 5 8 12 16 23 38 56 72 91

Enter Key:23

The key element is found at 6 position.

Complexity of Binary_Search:

The compexity of binary search is $O(\log n)$.

BASIC ALGORITHMS TO SORT ARRAY OF ELEMENTS

(BUBBLE, SELECTION AND INSERTION SORT):

Sorting is a process in which records are arranged in ascending or descending order. Sort method has great importance in real life problems.

Some Sorting techniques are

- 1) Bubble sort
- 2) Selection sort
- 3) Insertion sort

Bubble sort

The bubble sort is an example of exchange sort. In this method, repetitive comparison is performed among elements and essential swapping of elements is done. Bubble sort is commonly used in sorting algorithms. It is easy to understand but time consuming.

In this type, two successive elements are compared and swapping is done. Thus, step-by-step entire array elements are checked. It is different from the selection sort. Instead of searching the minimum element and then applying swapping, two records are swapped instantly upon noticing that they are not in order.

Example:

An array consists of [5, 1, 4, 2, 8].the array is sorted in ascending order

Original list	index	0	1	2	3	4
	ar	5	1	4	2	8

Pass 1:

5	1	4	2	8
Swap 5, 1 since $5 > 1$				

1	5	4	2	8
Swap 5, 4 since $5 > 4$				

1	4	5	2	8
Swap 5, 2 since $5 > 2$				

1	4	2	5	8
No swap since $5 < 8$				

Pass 2:

1	4	2	5	8
No swap since $1 < 4$				

1	4	2	5	8
Swap 4, 2 since $4 > 2$				

1	2	4	5	8
---	---	---	---	---

No swap since $4 < 5$

1	2	4	5	8
---	---	---	---	---

No swap since $5 < 8$

Pass 3:

1	2	4	5	8
---	---	---	---	---

No swap since $1 < 2$

1	2	4	5	8
---	---	---	---	---

No swap since $2 < 4$

1	2	4	5	8
---	---	---	---	---

No swap since $4 < 5$

1	2	4	5	8
---	---	---	---	---

No swap since $5 < 8$

Pass 4:

1	2	4	5	8
---	---	---	---	---

No swap since $1 < 2$

1	2	4	5	8
---	---	---	---	---

No swap since $2 < 4$

1	2	4	5	8
---	---	---	---	---

No swap since $4 < 5$

1	2	4	5	8
---	---	---	---	---

No swap since $5 < 8$

Algorithm:

Bubble_Sort (A [], N)

Step 1: Repeat for I = 0 to N - 1

 Begin

Step 2: Repeat for J = 0 to N - 1 - I

 Begin

Step 3: If (A [J] > A [J + 1])

 Swap (A [J], A [J + 1])

 End For

End For

Step 4: Exit

Program:

```
//Bubble sort
```

```
void main()
```

```
{
```

```

int ar[10],n,i,j,temp;
clrscr();
printf("Enter the No.Of elements:");
scanf("%d",&n);
printf("Enter %d elements:",n);
for(i=0;i<n;i++)
scanf("%d",&ar[i]);

//Bubble sort logic
for(i=0;i<n-1;i++)
for(j=0;j<n-1-i;j++)
{
if(ar[j]>ar[j+1])
{
temp=ar[j];
ar[j]=ar[j+1];
ar[j+1]=temp;
}
}

printf("\nThe sorted elements are:\n");
for(i=0;i<n;i++)
printf("%d\t",ar[i]);
getch();
}

```

Output:

Enter the No.Of elements:5

Enter 5 elements:5 1 4 2 8

The sorted elements are:

1 2 4 5 8

Complexity of Bubble_Sort

- The complexity of sorting algorithm is depends upon the number of comparisons that are made. Total comparisons in Bubble sort is

$$n(n-1)/2 \approx n^2 - n$$
- The worst case complexity for Bubble sort is $O(n^2)$ and best case is $O(n)$.

Selection sort:

The selection sort is nearly the same as exchange sort. Assume that we have a list on n elements. By applying selection sort, the first element is compared with all remaining (n-1) elements. The lower element is placed at the first location. Again, the second element is compared with remaining (n-1) elements. At the time of comparison, the smaller element is swapped with larger element. In this type, entire array is checked for smallest element and then swapping is done.

Example:

Original list

index	0	1	2	3	4
ar	9	4	7	5	8

Assume 1st element as min =9

Pass 1:

9	4	7	5	8
---	---	---	---	---

 Min=4 since $4 < \text{min}$

9	4	7	5	8
---	---	---	---	---

 Min= 4 only since $7 > \text{min}$

9	4	7	5	8
---	---	---	---	---

 Min= 4 only since $5 < \text{min}$

9	4	7	5	8
---	---	---	---	---

 Min= 4 only since $8 > \text{min}$

4	9	7	5	8
---	---	---	---	---

 Swap min(4) with 1st element

Assume 2nd element as min =9

Pass 2:

4	9	7	5	8
---	---	---	---	---

 Min= 7 since $7 < \text{min}$

4	9	7	5	8
---	---	---	---	---

 Min= 5 since $5 < \text{min}$

4	9	7	5	8
---	---	---	---	---

 Min= 5 only since $8 > \text{min}$

4	5	7	9	8
---	---	---	---	---

 Swap min(5) with 2nd element

Assume 3rd element as min =7

Pass 3:

4	5	7	9	8
---	---	---	---	---

 Min= 7 only since 9>min

4	5	7	9	8
---	---	---	---	---

 Min= 7 only since 8>min

4	5	7	9	8
---	---	---	---	---

 No swap

Assume 4th element as min =9

Pass 4:

4	5	7	9	8
---	---	---	---	---

 Min= 8 since 8<min

4	5	7	8	9
---	---	---	---	---

 Swap min(8) with 4th element

Algorithm:

Selection_Sort (A [], N)

Step 1 : Repeat for I = 0 to N – 2

Begin

Step 2 : Set MIN = I

Step 3 : Repeat for J = I + 1 to N-1

Begin

If A[J] < A [MIN]

Set MIN = J

End For

Step 5 : Swap A [I] with A [MIN]

End For

Step 6 : Exit

Program:

//Selection sort

void main()

{

int ar[10],n,i,j,min,temp;

clrscr();

printf("Enter the No.Of elements:");

scanf("%d",&n);


```

printf("Enter %d elements:",n);
for(i=0;i<n;i++)
scanf("%d",&ar[i]);
//Selection sort logic
for(i=0;i<n-1;i++)
{
min=i;
for(j=i+1;j<n;j++)
{
if(ar[j]<ar[min])
min=j;
}
temp=ar[i];
ar[i]=ar[min];
ar[min]=temp;
}

printf("\nThe sorted elements are:\n");
for(i=0;i<n;i++)
printf("%d\t",ar[i]);
getch();
}

```

Output:

Enter the No.Of elements:5

Enter 5 elements:9 4 7 5 8

The sorted elements are:

4 5 7 8 9

Complexity of Selection_Sort

- selection sort is a sorting algorithm, specifically an in-place comparison sort. It has $O(n^2)$ time complexity, making it inefficient on large lists, and generally performs worse than the similar insertion sort.

Insertion sort

In insertion sort the element is inserted at an appropriate place. For example, consider an array of n elements. In this type also swapping of elements is done without taking any temporary variable. The greater numbers are shifted towards end locations of the array and smaller are shifted at beginning of the array.

Example:

Original list

index	0	1	2	3	4
ar	9	4	7	5	8

Assume 2st element as temp =4

Pass 1:

9	4	7	5	8
---	---	---	---	---

9	9	7	5	8
---	---	---	---	---

Shift 9 right since 9>temp

4	9	7	5	8
---	---	---	---	---

Insert temp

Assume 3rd element as temp =7

Pass 2:

4	9	7	5	8
---	---	---	---	---

4	9	9	5	8
---	---	---	---	---

Shift 9 right since 9>temp

4	7	9	5	8
---	---	---	---	---

Insert temp since 4<temp

Assume 4th element as temp =5

Pass 3:

4	7	9	5	8
---	---	---	---	---

4	7	9	9	8
---	---	---	---	---

Shift 9 right since 9>temp

4	7	7	9	8
---	---	---	---	---

Shift 7 right since 7>temp

4	5	7	9	8
---	---	---	---	---

Insert temp since 4<temp

Assume 5th element as temp =8

Pass 4:

4	5	7	9	8
---	---	---	---	---

4	5	7	9	9
---	---	---	---	---

Shift 9 right since 9>temp

4	5	7	8	9
---	---	---	---	---

Insert temp since 7<8

Algorithm:**Insertion_Sort (A [], N)**

```
Step 1 : Repeat For I = 1 to N - 1
    Begin
Step 2 :   Set Temp = A [ I ]
Step 4 :   Repeat For J=I to J >= 0 AND A [ J-1 ] > Temp
    Begin
        Set A [J] = A [J-1]
        Set J = J - 1
    End For
Step 5 :   Set A [ J ] = Temp
    End For
Step 4 : Exit
```

Program:

```
//Insertion sort

void main()
{
int ar[10],n,i,j,temp;
clrscr();
printf("Enter the No.Of elements:");
scanf("%d",&n);
printf("Enter %d elements:",n);
for(i=0;i<n;i++)
scanf("%d",&ar[i]);

//Insertion sort logic
for(i=1;i<n;i++)
{
temp=ar[i];
for(j=i;j>0&&ar[j-1]>temp;j--)
{
ar[j]=ar[j-1];
}
ar[j]=temp;
}

printf("\nThe sorted elements are:\n");
for(i=0;i<n;i++)
printf("%d\t",ar[i]);
getch();
}
```

Output:

Enter the No.Of elements:5

Enter 5 elements:9 4 7 5 8

The sorted elements are:

4 5 7 8 9

Complexity of Insertion Sort

Best Case: $O(n)$

Average Case: $O(n^2)$

Worst Case: $O(n^2)$

SHORT ANSWERS ☺

1. Difference between selection sort and insertion sort.[May 2019]
2. Write an algorithm to find the maximum number in a given set.[May 2019]
3. How Binary search works?[June 2019]
4. How linear search is different from binary search?[June 2019]
5. How do you find the time complexity of bubble sort?[Aug 2019]
6. What is sorting and what is the importance of sorting?[Aug 2019]
7. Write an algorithm to find the given number is prime or not.[Dec 2018]
8. Find out no.of comparisons to sort {2, 1, 7, 4, 8, 6}[Dec 2018]

LONG ANSWERS ☺

ALGORITHMS

1. Devise an algorithm for linear search and explain with an illustration. [May 2019]
2. Write a c program to determine whether a given number is prime or not. [May 2019]
3. Explain the algorithm for finding the roots of the quadratic equation. [Aug 2019]
4. Define an algorithm to generate prime number series between m and n, where m and n are integers. [10 M] [Aug 2019]

SEARCHING

1. Apply linear search on { 22, 11, 66, 44, 99, 55, 88} [Dec 2018]

SORTING

1. What do you mean by sorting? Mention the different types of sorting. [Dec 2018]
2. Devise an algorithm for selection sort and explain with an illustration. [May 2019]
3. Give brief note on insertion sort with example. [June 2019]
4. Write a program in 'C' to print list of integers in ascending order using bubble sort and selection sort techniques. [10 M] [June 2019]
5. Write a program a program to sort array of integers using selection sort. [Aug 2019]
6. Apply bubble sort on { 22, 11, 66, 44, 99, 55, 88} [Dec 2018]
7. Explain quick sort in detail on {24, 12, 11, 76, 39, 12, 67, 34, 88, 91, 26, 45, 78} [Dec 2018]

ORDER OF COMPLEXITY

1. Give brief note on asymptotic notations. [May 2019]
2. Mention the complexity of linear search and binary search algorithms. [May 2019]
3. Discuss the time complexity of bubble sort. [June 2019]