PPS NOTES

programming & problem solving

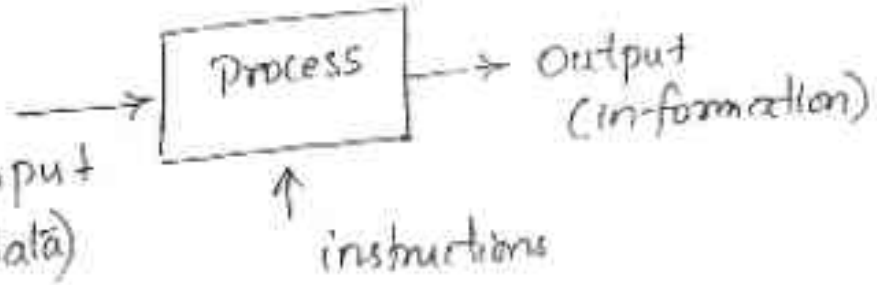
MJCET

writer: majusha

Unit-I

Introduction to computers

Computer :- A computer is an electronic machine which accepts data and instructions as input, processes the data and provides meaningful information known as output, which can be represented as,



Data :- raw facts are called as data.

Eg:- BE, Isem, CSE, online classes, 14 Dec, MJ COLLEGE, etc

Information :- Processed data is called as information.

Computer actually mean two things

i) Computer hardware (H/w) – which are the physical ware computers are made of.

ii) Computer software (S/w) – commands the H/w what to do and how to do.

Computer Hardware

Part of machine. It can only understand two basic concepts on (1) and off (0), which is called binary.

Computer Software

The H/w needs a S/w to instruct what has to be done.

A program is a set of instructions that is arranged in a sequence to guide a computer to find a solution for a given task.

The process of writing a program is called programming.

Classification of Computer S/w

i) System S/w ii) Application S/w

System S/w :

Provide a general programming environment in which programmers can create specific applications to suit their needs.

It acts as an interface b/w the H/w of the computer and the application S/w of the computer.

System S/w is designed to operate the computer H/w and to provide and maintain a platform for running application S/w.

Most widely used System S/w are computer BIOS and Device driver which provide basic functionality to operate and control the H/w connected to or built into computer.

BIOS - Basic Input Output System - built into the computer and is the first code run by the computer when it is switched on.

The key role of the BIOS is to load and start the operating system. The first function that BIOS performs is to initialize and identify system devices like keyboard, mouse, [illegible] and other info.

The code in the BIOS chip runs a start up test called POST (Power On Self Test) to ensure that the system devices are working correctly.

BIOS is stored on a ROM chip built into computer.

Application software

Designed to solve a particular problem for users.

Programming Languages

The operation of a computer is controlled by a set of instructions (Programs) which tells the computer

1) what operations to perform
2) where to locate data
3) how to present results and make decisions

The language used for communicating with a computer is known as the programming language.

Programming languages are of different [illegible]

1) Machine language } low level
2) Assembly language }
3) High level language - procedure oriented
- problem oriented
- natural lang.

Instructions are designed by using a [illegible] code which consists of 2 fields: the operation code (opcode) and address part which are to be represented as,

opcode	address

[illegible] used to specify operation part [illegible] to refer to specific address

Eg:- for add

To execute the above instruction, we need to fetch the values of operands stored in memory at some address specified in address part of the instruction and then perform the operation specified in op code part of instruction.

+	a, b

Machine language

Only lang. that the computer understand. All the commands and data values are represented using 0's and 1's, corresponding to the off and on electrical states in a computer.

The main advantage of m/c lang is that the code can run very fast and efficiently since it is directly executed by the processor.

The m/c lang is difficult to learn and is far more difficult to edit if errors occur. The code written in m/c lang is not portable across systems.

Assembly language

It uses symbolic notation to represent machine instructions. The symbolic codes used are known as mnemonics and are easy to write and remember, [illegible] abbreviations rather than numbers in binary.

Eg: ADD, SUB, MUL, DIV, CMP, LOAD, STORE etc.

The code written in assembly lang will need a translator known as the assembler to convert symbolic [illegible] instructions into machine lang (binary).

High level language

In this programs are written in English-like manner, making them more convenient to use and giving the programmer more time to address a problem.

A translator is required to translate the instructions written in high level lang into computer executable m/c lang. [illegible] compiler and interpreter.

Note: [illegible] lang is used and [illegible] such as [illegible]

Procedure oriented lang

It follows a step by step approach to solve a task through a sequence of instructions.

Eg: C

Problem-oriented lang

No standard rule to define [illegible] concentrate on what instead of how. Mostly [illegible] in English like [illegible] i.e.

Eg: SQL – query lang.

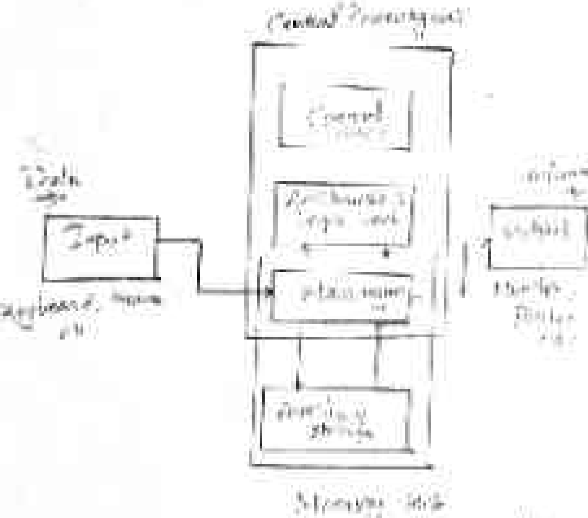
Natural lang

Centred on solving problems using conditions given to the program.

Eg: Robotics, Neural N/w

Block diagram of a computer

The working process of a computer from inputting the data to generating the desired results can be represented using block diagram as:



Input Unit: Consists of input devices which are used to input information (or) instructions into the computer system.

Two major functions of input unit are:

a) It converts the inputted data (or) instructions into binary form for further processing.

b) It transmits the data to the main memory of the computer.

Central Processing Unit (CPU)

CPU is known as the brain of the computer. It is an electronic hardware that carries out the instructions ... the operations (arithmetic and logical) and the calculations, are performed inside the CPU.

(The control unit (CU) and arithmetic logic unit (ALU) are jointly known as the CPU)

(The control unit of a CPU controls all the activities and operations of the computer. It is also responsible for controlling input/output [illegible] and other devices connected to the computer.) The control unit determines the sequence in which the given instructions are executed.

(The ALU performs arithmetic and logical operations, it also performs logical operations such as comparing, selecting, and merging data)

(The information or data is transmitted to ALU from the storage unit only when it is needed.)

Memory Unit

Used to store data and instruction before and after processing. The memory unit transmits the information to other units of the computer system when required.

We have two types of memory unit i.e. "primary memory and secondary memory.

Primary memory (Main / [illegible])

Available as a built in unit of a computer. The primary memory is represented as a set of locations with each location occupying exactly 8 bits (1 byte). Each byte in the memory is identified by a unique address and the data is stored in the binary form in these memory locations.

Commonly used primary memories are –

ROM (Read only Memory)

Stores data and instructions even when the computer is turned off. It is the permanent memory of the computer where the content cannot be modified by user.

ROM is a chip which is attached with mother board (used for connecting circuits and input/output devices). ROM is generally used to store the Input/output system control or BIOS which perform POST.

RAM (Random Access Memory) :-

RAM is the read/write unit in which the information is retained only as long as there is power supply. The data will be lost if they are disconnected from the power supply. RAM is volatile memory that temporarily stores data and applications as long as they are in use.

Secondary memory (Auxiliary / Permanent memory)

It is used to store data permanently. It is not possible to store the files using primary memory, so to store the data permanently for future use, we use secondary memory.

Eg:- Hard disk

Note :- Primary memory is the only memory that is directly accessible to the CPU.

Output Unit

It consists of devices that are used to display the results of computer processing.

Eg:- Monitor, Printer etc.

It accepts the data or information in binary form from the main memory and converts the binary data into human readable form.

Algorithm and Flow Chart

A typical programming task can be divided into two phases

i) Problem solving phase :- In this phase, we produce an ordered sequence of steps that describe solution of a problem. [illegible] algorithm. Algorithm is a [illegible] and instructions in sequence to solve the problem

Implementation phase - [illegible]

Characteristics / Features - Algorithm

i) Input and output should be defined precisely

ii) Each step in algorithm should be clear and unambiguous

eg:- [illegible]

iii) Algorithm should be effective and should complete in a finite time

iv) [illegible]

Eg:- Add two no.

step 1 : Start

step 2 : Declare 3 variables [illegible]

step 3 : Read values for [illegible]

step 4 : Add [illegible] and [illegible] the result to sum [illegible]

step 5 : Display sum

step 6 : stop.

(3)

Find max among 2 no.

step 1 : start

step 2 : Declare 2 variables a and b

step 3 : Read values for a and b

step 4 : Check if a > b

display a is max no

else display b is max no

step 5 : stop.

Advantages of algorithm

i) as it is a step wise representation of a solution, makes it easy to understand

ii) uses a definite procedure

iii) [illegible] logical sequence

iv) easy to debug

Disadvantage

i) takes time

ii) algorithm [illegible] a complex program

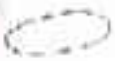
Flow Chart

A flowchart is a graphical representation of workflow or process. It shows steps in sequential order and is widely used in presenting the flow of algorithm, workflow or process

Note - Pseudocode : informal way which helps programmer develop algorithm.

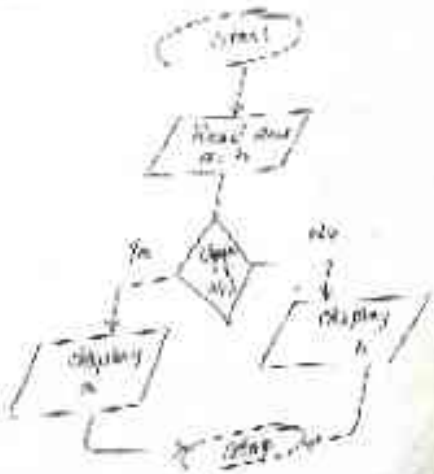
A flow chart is a diagrammatic representation of an algorithm. A flow chart can be useful for both writing programs and explaining the program.

Symbols used in flow chart

Symbol	Purpose	Description
	Terminal (start/stop)	Represents the start and end of a flow chart
	Input/Output	Used for input and output operations
	Processing	Used for arithmetic operations and data manipulations
	Decision	Used to ensure that the program follows the correct path
	Flowline	Indicates the flow of logic by connecting symbols
	Connector	Used to join different flowlines

flow chart to add two

flow chart to find max among two



Advantage of Flow chart

1) Convenient method of communication
2) A key to correct programming
3) Indicate the task played at each level
4) Facilitates troubleshooting
5) Promote logical accuracy

Disadvantage of Flow chart

1) Waste of time and slows down the process of s/w development
2) Sometimes the complex logic of the program is quite complicated to draw
3) Modification or alteration of the process is very hard

Computing Environment

It is a collection of computer which are used to process and exchange information to solve various types of computing problem.

Types of Computing Environment

1. Personal Computing Environment
2. Time sharing computing environment
3. Client-Server computing environment
4. Distributed computing environment
5. Grid computing environment
6. Cluster computing environment

Personal Computing

It is a stand-alone machine. The complete program resides on stand-alone machine and executed from the same machine.

Eg:- laptops, mobiles, printer, computer system at home.

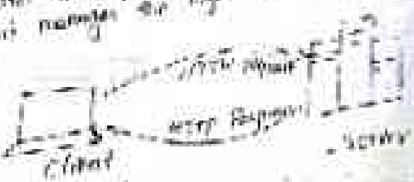
Time Sharing Computing

It is a stand-alone computer in which a single user can perform multiple operations at a time by using a multitasking operating system. The processor time is divided among different tasks which is called Time-sharing.

Eg:- We can listen to music while writing something in a text editor.

Client-Server Computing

It contains two machines (client & server) which exchange the information through an application. Here client is a normal computer like PC, tablet, mobile etc. and server is a powerful computer which stores large volume of files and manages the files.



Distributed Computing

The complete functionality of the s/w is not on a single computer but is distributed among multiple computers. These computers communicate with each other and is able to perform the complete task.

Grid Computing

It is a collection of computers from different locations which work for a common goal.

Cluster Computing

It is a collection of interconnected computers which work together to solve a single problem.

Program Development Life Cycle

The set of steps or phases used to develop a program in any programming lang. and contains the following phases:-

1) Problem definition – define the problem, what is our requirements, what should be the output of the problem.
2) Problem Analysis – the requirement to solve the problem – gathering the requirement necessary to solve the problem.
3) Algorithm Development – develop step-by-step procedure to solve the problem.
4) Coding and Documentation – convert the algorithm into actual programming instructions to write the [illegible] defined in algorithm.
5) Testing & Debugging – check whether the code written is solving the specified problem or not.
6) Maintenance – the program is actively used by the users.

Language Processor

A computer understands instructions in machine code i.e. in form of 0s and 1s.

The source code cannot be executed directly by the computer and must be converted into machine lang to be executed.

A special translator system software is used to translate the program written in high level lang into machine code. The program after translated into machine code is called object code.

Three types of lang processors are:

1) Assembler – used to translate the program written in assembly lang into machine code.

Source code (Assembly lang) → [Assembler] → Object code (machine lang).

2) Compiler – read the complete source program written in high level lang as a whole and translate it into machine lang.

Eg: C, C++, Java

Source code (high level lang) → [Compiler] → Object code (machine lang)

Interpreter: The translation of single statement of source program into machine code is done and executes it immediately before moving on to the next line is called interpreter.

The interpreter moves on to the next line only after removal of the error.

Eg. Python, Matlab

Compiler	Interpreter
1) Scans the entire program and translates it as a whole into machine code	Translates program one statement at a time
2) The overall execution time is comparatively faster than interpreter	The overall time of execution is slower than compiler
3) Generates intermediate code which further requires linking, requires more memory	No intermediate code generated, memory efficient
Eg. C, C++, Java etc.	Eg. Python, [illegible], Matlab etc.

Computing Concepts

We can communicate with the computer using electric signals which are represented as a series of pulses (1) and no pulse (0) called binary digits (bits), abbreviated as binary digits.

A series of 1s and 0s are used to represent characters, numbers (digits), special symbols and also provide a way for humans and computers to communicate with one another.

The values represented by binary digits (0&1) are known as binary numbers.

Binary Number System

It is a numerical number system which represents numeric values using two digits 0 and 1 (bits). 2 is the base (radix) of binary number system. The internal functioning of a computer system is carried out in binary number format, we can say the used binary codes & number systems are used in electronic systems are

i) Binary Coded Decimal (BCD)

ii) American Standard Code for Information Interchange (ASCII).

Binary Coded Decimal (BCD)

1) Any decimal digit in binary is represented &
   represented in powers of 2.
2) We represent a binary no. either by using
   4-bits representation called as nibble or
   by using 8-bits representation called as
   byte (word).
3) A binary number is read from right to
   left.

Decimal	Binary 2^3 8	2^2 4	2^1 2	2^0 1
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	1	0	1	0
11	1	0	1	1
12	1	1	0	0
13	1	1	0	1
14	1	1	1	0
15	1	1	1	1

Note -

1) We can represent 16 no. using 4 bits representation in binary from 0 to 15.
2) The maximum no. which can be represented in binary using 4-bits representation is 15.

Eg:- To represent 16 in binary, we require 5-bits as
2^4 2^3 2^2 2^1 2^0
1 0 0 0 0
16 8 4 2 1

but there is no number in binary which is represented in binary using 5 bits (16 to 31), so we fill up the remaining 3 bits from the left with zeros.

So the most appropriate representation of 16 in binary is,
00010000

Binary Arithmetic

Addition rule:

0 + 0 → 0
0 + 1 → 1
1 + 0 → 1
1 + 1 → 0, and a carry 1 is forwarded to next left position

Eg: 3 → 0011
+ 2 → 0010
5 → 0101

8 → 1000
+ 4 → 0100
12 → 1100

9 → 00001001
+ 9 → 00001001
18 → 00010010

Note -
i) There is no negative number representation in binary.
ii) We represent negative number in binary using the complement.

2's Complement:
Adding one to 1's complement is 2's complement.

(4)

1's Complement:
1's complement of 0 is 1 and 1 is 0.

Eg: 3 ⇒ 2 ⇒ 00000010
↓ 1's comp
11111101
+1
11111110
↓ 1's comp
00000001
+1
00000010 (-2)

3 ⇒ 00000011
-2 ⇒ 11111110
1 00000001

- 6
13
-2

6 ⇒ 00000110
↓ 1's comp
11111001
+1
11111010 (-6)

-6 ⇒ 11111010
4 ⇒ 00000100
11111110 (-2)
-2

Note - After performing the arithmetic, the answer will be in the correct format.

Binary numbers can be represented many way :

i) Signed number representation
ii) Unsigned number representation

Signed number representation

While representing a binary no. using 8 bits notation, we distinguish positive and negative numbers using signed notation.

― ― ― ― ― ― ― ― ← Lo or right most bit (least significant bit) (LSB)

↑ left most bit (Most significant bit) (MSB)

In a positive number, the MSB value is 0 and in negative number the value is 1. The MSB is considered as the sign bit.

Eg:- 2 → 0 0 0 0 0 0 1 0 (MSB)

−2 (8 bit) → 1 1 1 1 1 1 1 0 (MSB)

Unsigned number representation

In this notation, there is no sign bit. It only represents positive numbers, i.e. all binary represent ~~are~~ positive numbers only.

Eg:- $(132)_{10} = (\quad)_2$

2^7 2^6 2^5 2^4 2^3 2^2 2^1 2^0
128 64 32 16 8 4 2 1
1 0 0 0 0 1 0 0

$(132)_{10} = (10000100)_2$

ASCII :-

ASCII is a standard alpha numeric code which is used to represent alphabets, numbers and special symbols using the seven bit code format.

Eg:- ASCII value of A = 65, B = 66, C = 67
ASCII value of a = 97, b = 98, c = 99
ASCII value of zero = 48, one = 49

Note:- The ASCII characters are a set of 128 (0 to 127) (128 decimal values) which are assigned to alphabets, numbers and special symbols.

Octal Number System

This number system uses 3 bits to represented a no. and the numbers are also created in powers of 2. The base of the octal no. system is 8

Decimal	Octal 2^2 2^1 2^0 (4 2 1)
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

Hexadecimal Number System

It was used to represent a no. in binary representation. 4 binary digits are used to represent. 0 to 9 and from 10 to 15 represented by A to F. The base of the hexadecimal number system is 16

Decimal	Hexadecimal 2^3 2^2 2^1 2^0 (8 4 2 1)
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
11	B
12	C
13	D
14	E
15	F

Conversions

Decimal to binary

we have two methods are:

① standard method – according to the decimal number expand the digits in powers of 2 by using 8 4 2 1 representation.

Eg: $(76)_{10} = (\ )_2$

$2^6\ 2^5\ 2^4\ 2^3\ 2^2\ 2^1\ 2^0$

1 0 0 1 1 0 0 ⇒ $(1001100)_2$

② Double Dabble method

In this method we divide the decimal number repeatedly by 2 while noting the remainder values from bottom to top.

Eg: $(76)_{10} = (\ )_2$

$(1001100)_2$

2 | 76
2 | 38 – 0
2 | 19 – 0
2 | 9 – 1
2 | 4 – 1
2 | 2 – 0
  1 – 0

1001100

Binary to decimal

$(1101)_2 = (\ )_{10}$?

$1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$

$= 1 \times 8 + 1 \times 4 + 0 \times 2 + 1 \times 1$

$= 8 + 4 + 0 + 1 = 13$

$(1101)_2 = (13)_{10}$

decimal to octal:

divide the decimal number repeatedly by 8 noting down the remainder from bottom to top

Eg: $(126)_{10} = (\ )_8$?

8 | 126
8 | 15 – 6
  1 – 7

$(126)_{10} = (176)_8$

octal to decimal

$(176)_8 = (\ )_{10}$?

$= 1 \times 8^2 + 7 \times 8^1 + 6 \times 8^0$

$= 1 \times 64 + 7 \times 8 + 6 \times 1$

$= 64 + 56 + 6$

$= 126$

$(176)_8 = (126)_{10}$

Binary to octal:

Consider the given binary number from right side and divide it into 3 bits each

Eg: $(1101111001110)_2 = (\ )_8$?

1 | 101 | 111 | 001 | 110
1 | 5 | 7 | 1 | 6

$(1101111001110)_2 = (15716)_8$

octal to binary: Convert the octal to binary with each of 3 bits each in binary. Read each no. of octal using 3 bits each ... (421) ...

Eg: $(156)_8 = (\ )_2$?

$= (001101110)_2$

Decimal to hexadecimal

we divide the decimal no. repeatedly with 16, noting down the remainders from bottom to top.

Eg: $(159)_{10} = (\ )_{16}$?

$16 | 159$ — $9 - 15(F)$

$(159)_{10} = (9F)_{16}$

Hexadecimal to decimal

Eg: $(9F)_{16} = (\ )_{10}$?

$= 9 \times 16^1 + F \times 16^0$

$= 9 \times 16 + 15 \times 1$

$= 144 + 15 = 159$

Binary to hexadecimal — divide the binary into 4 bits each from left to right

Eg: $(1111011000011)_2 = (\ )_{16}$?

$1\ 1110\ 1100\ 0011 = (1EC3)_{16}$

Hexadecimal to binary — represent each no. using 4 bit each

Eg: $(9F)_{16} = (\ )_2$?

$(9F)_{16} = (10011111)_2$

Introduction to C language

C is a structured programming lang. developed at AT & T Bell Laboratories designed by Dennis Ritche in 1972.

C was primarily used for programming an operating system called UNIX.

Steps in learning C language

Steps in learning C lang. are as follows:

Alphabets → Constants → Keywords → Instructions → program
(characters, digits & special symbols) (Variables & Datatypes)

C language character set

C language character set are as follows:

(i) Source characters: This consist of alphabets, digit and some special symbols, which can be combined to form variables.

Eg: Alphabets — A to Z in upper case, a to z in lower case

Digits — 0 to 9.

Special symbols — #, $, %, &, * etc.

(B) Execution Character Set

This comes only at the time of execution. Each character has a backslash (\) before it which indicates that it is an execution character set.

Execution char	Meaning
\n	next line (newline)
\b	backspace (1 space)
\v	vertical tab
\t	horizontal tab
\a	backspace
\\	prints double quote
\"	prints single quote
\'	

Note:-
The execution character set is also called as Escape Sequence. These by placing a backslash in front of a particular character are changing the original meaning of displaying the character on the monitor. ie the character is not displayed on the screen. Only get interpreted according to the character after backslash at the time of execution.

Uses of C language

i) C language is used for system programming i.e. used to implement operating systems and embedded system applications.

ii) Compilers, interpreters and libraries of most of the programming languages are often implemented in C.

iii) C is sometimes used as an intermediate language for implementing other languages.

iv) To implement end user applications.

Structure of C program

C program is composed of preprocessor commands, and one or more functions [illegible] syntax;

```
/* Comment (s) */ [illegible] of
preprocessor directive
main( )
{
        declarative part;
        stmt1;
        stmt2;
        stmt3;
        ===
        stmtn;
}
```

The execution of a C program begins at main() i.e every C program gets executed by executing main(). It is the compulsory function in every C program.

• Creating and Executing C programs

We have 4 steps in creating and running C program as,

i) Writing & editing the Program

Source code is entered/written through a text editor, with .c extension and it should be translated into machine language code with help of a compiler.

ii) Compiling the Program

This is carried out by the translator compiler which translates the source code to object code with .obj extension [illegible] error free.

iii) Linking the program with standard libraries

Linking the program is required for having the object code with required library module i.e this links all the built-in library function definitions with the object code for making executable with .exe as its extension.

iv) Running / Executing the Program

The executable code with .exe extension is loaded into main memory with help of a loader and the execution of a program begins.

All the above 4 steps can be represented as,

Text editor
(source code-english)
.c

↓

Compiling the source code
(object code (.obj) in binary)
incomplete code

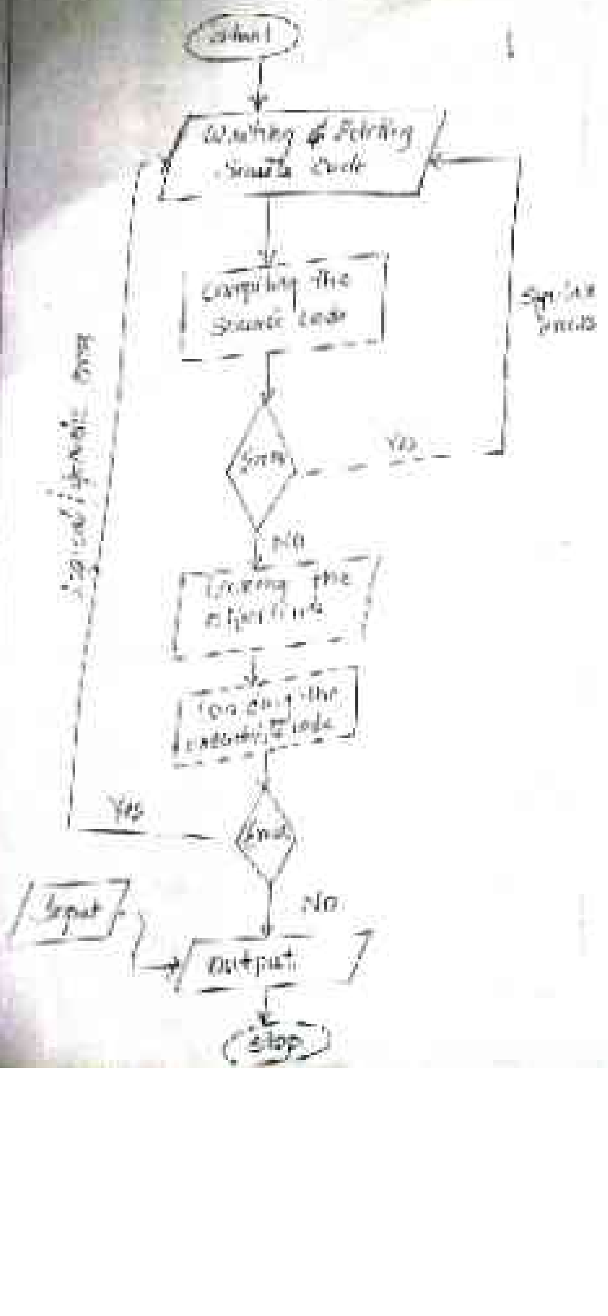
↓

Linking the object code
(executable code (.exe) in binary)
complete code

↓

Running / loader
(ready for execution).

The whole process can be represented in form of a flow chart as,



Files used in a C program

Every C program has four kinds of files associated with it are,

1) Source Code file (.c) :- This file contains the source code which is in English like language. The file extension of any C source code file is .c

2) Header file (.h) :- Allows the inclusion of appropriate header file in the C program with precompiled declarations so that even though the program is compiled the linker would give us the definitions of the library functions used in the program.

eg. <stdio.h>, <conio.h>, <string.h>

3) Object file (.obj) :- Object files are generated by the compiler as a result of processing the source code file. Object files contain compact binary code, which cannot be executed as it is an incomplete program, without the library function definitions.

4) Executable file (.exe) :- The linker links the object code with various library module definitions used in the program to produce a binary executable file, ready for execution.

Tokens

In every C program the basic element recognised by the compiler is a single character or group of characters called as tokens. Tokens are the basic elements of the C lang.

C tokens can be defined as:

1) Identifiers
2) Keywords / Reserved words
3) Constants
4) Operators
5) Separators

Identifiers:-

Identifiers are the names given to the programming elements like variables, arrays, functions, labels etc.

Identifiers are used to identify a particular element in a program [illegible].

Rules

1) It should start with a letter & underscore [illegible]
2) Identifiers are case sensitive [illegible]
3) It can be of any length [illegible]

Keywords / Reserved words

Keywords have fixed meaning and the meaning cannot be changed. There are 32 keywords in C which are written in lowercase letters.

eg: int, float, char, double, signed, unsigned, if, else, if-else, switch, break, continue etc.

Rules:

[illegible]

Constants

Constants in C refer to fixed values that do not change during the execution of the program.

Constants
- Numeric constants
  - Integer constants
    - decimal
    - octal
    - hexadecimal
  - Real constants
    - single precision
    - double precision
- Character constants
  - single char const
  - string const

Integer constants :- An integer constant refers to a sequence of digits with no decimal point. Integer constants can be specified in 3 ways:

1) Decimal - represented by 0 to 9 digits [illegible]
2) Octal - An integer beginning with zero (0) is considered as octal number and contains 0-7 digits only.
3) Hexadecimal :- An integer beginning with 0x is considered as an hexadecimal number which contains 16 digits.

eg: zero in decimal - 0 → octal
zero in octal - 00
zero in hexadecimal - 0x0 → hexadecimal

Rules

1) It must have atleast one digit
2) It may be +ve/-ve
3) It should not have decimal point
4) No commas and spaces are allowed in expressing integer constant

Real constants (or) floating point constant

Real constants are written in two forms as,

1) fractional form

It consists of a series of digits representing the whole part of a number, followed by a decimal point, which is followed by a series of digits representing the fractional part

Eg: 15.125, -57.2708 etc.

Rules

1) It may be +ve/-ve
2) Must have decimal point
3) No commas and space allowed

Exponential form

In this, the real constant is represented in two parts: mantissa part and exponent part

Eg: 0.000534 can be represented in exponential form as 5.34e-4

like wise 43000 as 4.3e4

Rules

1) Mantissa and exponent is separated by e.
2) Mantissa and exponent may be +ve/-ve

Character constant

Single character constant - contains a single character enclosed in a pair of single quotes — 'a'

String constant - consists of sequence of characters enclosed in double quotes

Eg: "CSE", "HELLO" etc.

Operators - two types of operators

1) Single operator: all the arithmetic operators
2) Compound operator: as the input and relational operators

Separators: ; , ( ) { } [ ] " " etc.

Eg:

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

Variables

A variable is a storage place which has some memory allocated to it. Variable is used to store some form of data. Different types of variables require different amount of memory.

Syntax :-

datatype var1, var2, [illegible];

Eg:-

int a; => 2 bytes

float b; => 4 bytes

char c; => 1 byte

Variable assignment

A variable assignment is a process of assigning a value to a variable.

Eg:

int a = 10;

float b = 2.5;

Type qualifiers

The keywords which are used to modify the properties of a variable are called type qualifiers.

We have two types qualifiers in [illegible]

1) const qualifier :- this is used to declare a variable to be read-only (constant). Its value is not changed.

Syntax

const datatype var = value;

(or)

datatype const var = value;

Eg:- const float pi = 3.14;

(2) Volatile qualifier :- This is used to declare a variable that can be changed explicitly.

Syntax :

volatile datatype var;

(or)

datatype volatile var;

Datatypes :-

Datatypes are declarations for variables. This decides the type of values which can be processed by the language [illegible] variable within the [illegible]

Datatypes

- Basic / Built in / primary / primitive datatypes
- user defined data types
- Derived datatypes

Primary datatypes

These are fundamental data types in C. Every C compiler supports four primary data types as,

- int — used to denote an integer value
- char — used to denote a character type
- float — } used to denote a floating point
- double — } type (real no).

Basic datatype

- integer
  - signed
  - unsigned
  - int (-32,768 to 32,767)
  - short int [illegible]
  - long int (-2,147,483,648 to 2,147,483,647)
  - int [illegible]
  - short int [illegible]
  - long int [illegible]
- Floating point
  - float [illegible]
  - double [illegible]
  - long double [illegible]
- Character
  - char [illegible]
  - signed char [illegible]
  - unsigned char [illegible]

Note: void datatype — nothing or no value. used to specify function cannot return any thing

User defined datatypes

The user can defined new datatypes which are equivalent to existing datatypes. These user defined datatypes can be used later to declare variables of user choice.

Eg: typedefinition, structure, union, enumeration.

type definition

The keyword typedef is used to declare new datatype names in C.

Syntax: typedef datatype identifiername;

Eg: typedef int age;
age male, female;
↳ user defined datatype name

typedef char age;
age m1;
↳ user defined datatype name

typedef float salary;
salary floating, working;
↳ user defined datatype name.

Enumeration data type (enum)

It is a user defined data type in C, which is used to assign names to integral constants. The keyword "enum" is used to declare an enumeration data type and [illegible].

Syntax :-

enum identifier { enumeration list };

↓ keyword ↓ [illegible]

Eg:
```
#include <stdio.h>
main( )
{
    enum week { Mon, Tue, Wed, Thu, Fri, Sat, Sun };
    enum week day;
    day = Wed;
    printf("%d", day);
}
```

Note :- The compiler automatically assigns the enumeration list with [illegible] values in [illegible] order, example, [illegible] from 0 to 6.

So day = Wed returns 2.

Operators & Expressions

An operator is a special symbol or character which is used to perform certain mathematical (or) logical operations.

The data items that operators act upon are called operands.

An expression is a combination of both operands and operators written in any form as per the syntax of C lang.

The different types of operators used in C lang are as follows:

1) Arithmetic Operators / Binary operators :- which are used to perform mathematical calculations.

operator	example
+ (addition)	a + b
- (subtraction)	a - b
* (multiplication)	a * b
/ (division)	a / b
% (modulus)	a % b

Note :- The modulus (%) operator returns the remainder as an answer.

Eg: $10 \% 2 = 0$, $15 \% 5 = 0$, $21 \% 2 = 1$

[illegible]

3) 1%10 = 1 , 23%7 = 2 , 5%10 = 6 etc.

If numerator is less than denominator while using modulus operator, the numerator itself is returned as an answer.

4) Modulus only works for integer (short, int) not with float / double.

5) Unary operator

Unary operator is an operator which operates upon a single operand to produce a new value.

Operator	Meaning
unary minus (-)	negates the value of operand
unary plus (+)	changes the sign of only negative operand
increment op (++)	increments the value of operand by 1
decrement op (--)	decrements the value of operand by 1
logical negation (!)	reverse the logical state (if true → false)
address operator (&)	gives the address of variable in memory
sizeof()	returns the size of types in bytes

Logical negation (!)

used to reverse the logical state of operand — true becomes false and false becomes true.

c = 5,
!(c == 5)
!(5 == 5)
!(1) = 0.

Address operator (&)

gives the address of the variable (memory location).

[a = name, 10 = content, 5352 = address (ram)]

int a = 10;
printf("\n the address location that value of a = %d", &a);
value [illegible] address location which value [illegible]
(= 10)

sizeof() operator

gives the size of variable in bytes

e.g. int a; => sizeof(a) => 2 bytes
float b; => sizeof(b) => 4 bytes
char ch; => sizeof(ch) => 1 byte.

Assignment operator (=)

used to assign a value from right side operand to left side operand.

identifier = expression;

Eg: c = a+b;

Here the expression on right side is evaluated and assigned to the variable using equal (=) operator.

Note:

Short hand assignment operators

[illegible]

Relational operator

used to compare the relation between two numbers. If the relation is true, it returns 1 or else 0.

Operator	Meaning	Example
>	greater than	[illegible] return 1
<	less than	[illegible] return 1
>=	greater than or equal to	[illegible] return 1
<=	less than or equal to	[illegible] return 0
==	Equal to (for comparison)	[illegible] return 0
!=	not equal to	[illegible] return 1

Logical operator

used to combine two or more conditions. They are commonly used in decision making in C programming. Evaluating expressions the result is [illegible] in logical expression.

Operator	Meaning	Example
&&	logical AND	[illegible]
||	logical OR	[illegible]
!	logical NOT	[illegible]

Conditional operator (?:)

Conditional operators return a value if condition is true and returns another value if condition is false.

Syntax -

Variable = (condition) ? true value : false value;

(or)

Condition ? (expression1 ? expression2 : expression3);

Eg: a = 5, b = 3, max;

max = (a > b) ? a : b;

= 5 > 3 ? 5 : 3

true

max = 5.

x = 10, y = [illegible] 20

[illegible]

y = 10

Note: According to the condition of the condition either expression2 (true value) is evaluated or expression3 (false value) is evaluated as an answer.

Comma operator (,)

In C, the comma (,) is used as an separator and an operator.

Eg: int a, b, c;

← used as separator

int a[5] = {1, 2, 3, 4, 5};

Bitwise Operator

Used for manipulating data at the bit level. They are used to [illegible] the [illegible] -ming process -[illegible]

Operator	Meaning
~	Bitwise complement
&	Bitwise AND
|	Bitwise OR
^	Bitwise EX-OR
<<	Left shift
>>	Right shift

Bitwise complement (~) - changes 1 into 0 & 0 into 1.

Eg: a = 35 : 00100011

~a => 11011100

Bitwise AND, OR, EXOR (&, |, ^)

Eg: a = 12 => 00001100

b = 25 => 00011001

a & b => 00001000

a | b => 00011101

a ^ b => 00010101

Left shift (<<)

The left shift operator will shift the number by one bit to the left and the [illegible] significant bits in the number will be [illegible] and 0 will the value is gifted on right side.

Eg: x = 7 => [illegible]

= 00000111 → only [illegible]

So add a bit at right side of [illegible]

x<< = 00001110

Right shift (>>)

The right shift operator will shift the number by one bit position to right and the right most bits in the number will be [illegible] and 0 is [illegible] on left side

Eg: x = [illegible] => [illegible]

= 00001[illegible]

So add a bit at left side [illegible] value

x>> = 0000[illegible]

Note: left shift increase in value and in right shift decrease in value

Precedence & Associativity

Precedence is the priority for grouping different types of operators with their operands & [illegible] the order of evaluation of operators in an expression.

Associativity: left-right / right-left order for group operands to operators that have the same [illegible] operands to define the order in which operators [illegible] are evaluated in an expression [illegible] of the same precedence of the expression.

Eg: *, /, % – first
+, – – second

Eg: 1) [illegible] + [illegible] × 6 — (1)
180
[illegible] = [illegible] — (2)
[illegible]

2) 34 + 12/4 – 45 — (1)
— (2)
34 + 3
37 — 45
= –8

Practice

i. [illegible]

j. [illegible]

k. [illegible]

Console Input/output function

Keyboard and screen together are called console. These function receive input from keyboard and write them on VDU (visual display unit). These are classified as

a) Formatted Input/Output function
b) Unformatted Input/Output function

Formatted Input/Output function (scanf(), printf())

scanf() – used to read/input values of variables using the standard input device keyboard.

Syntax:

scanf("format string", &v1, &v2, &v3 ..., &vn);

Format string represents the format specification. The symbol & (ampersand) represents the memory address where the variable value is to be stored.

int
- %d – decimal
- %o – octal
- %x – hexadecimal
- %u – unsigned int value (no sign) (+ve)
- %i – int value in any form decimal/octal/hexadecimal

float
- %f – real value up to 6 digits precision
- %e – exponential form

char
- %c – single character
- %s – string

printf() – used to print/display values of variables using standard output device – monitor.

Syntax:

printf("control string to be displayed", variables/names);

Note: If the character read by printf() is % or \, it does not print it but reads the next character and interprets it having special meaning as it.

Unformatted Input/Output function

Unformatted input and output function only work with character data type. These function doesn't require any format specifiers, as they work with character datatype.

Character Input/Output function

getchar() :- reads one character at a time.

Syntax :-

identifier = getchar();

Eg :- char ch;

ch = getchar(); [illegible]

[illegible]

putchar() :- print only one character at a time

Syntax :- putchar(variable);

Eg char ch;

ch = getchar();

putchar(ch); [illegible]

gets() :- reads a single string at a time.

Syntax. gets(variable);

Eg :- char string[20];

gets(string); [illegible]

puts() :- prints the string variable

Syntax :-

puts(variable);

Eg :- char string[20];

gets(string);

puts(string);

Type casting :-

Type casting is converting one data type into another one. It is also called data conversion or type conversion [illegible]

Ex :-

1. Implicit type casting [illegible] Implicit type casting is done by compiler automatically without any [illegible] [illegible] This is also called as assignment operator [illegible] standard type conversion

Syntax - (variable) = expression;

Eg: int a = 5, b = 2;

float c;

c = a/b;

c = 5/2 = 2.

Here as the numerator is an integer constant we get only 2 as an answer instead of 2.5 i.e. truncate the precision part.

we can only have approximate answers but not accurate value using implicit casting ... correct answer.

2) Explicit type casting (Forced casting)

Used to convert one type of value into another without disturbing the contents of values. The operator used is called casting operator and the process is called casting.

Syntax - (variable) = (datatype) expression

Eg: int a = 5, b = 2;

float c;

c = (float) a/b;

c = (float) 5/2 = 2.5

Control structures in C

Flow of control through any given function is implemented with three basic types of control structures.

1) Sequential - default mode - sequential execution of code.
2) Selection / Branching - used for decisions i.e. choosing b/w two or more alternative paths.
3) Repetition / Iteration - used for looping i.e. repeating a piece of code multiple times.

Branching / Selection control statements - Branching is deciding what actions to take place ... flow of execution ... selection. C lang supports 2 types of selection / branching statements.

- Branching / selection stmts
  - Conditional stmts
    - Simple if stmt
    - if-else stmt
    - if-else-if stmt
      - if-else ladder
      - special if stmts
    - switch stmt
  - Unconditional stmts
    - break
    - continue
    - goto

Conditional control structure

1) Simple if statement

Syntax:

```
if (test condition)
{
    statement;
    statement;
    ==========
    statement;
}
```

The statements inside the if block get executed only if the test condition is true.

Flowchart:

Ex: Eligible to vote

```
main()
{
    int age;
    printf("Enter age:");
    scanf("%d", &age);
    if (age >= 18)
    printf("You are eligible to vote");
```

Note: In simple if statement, when the condition is false, the user is clueless regarding what happens.

if-else statement:

An if else part can be followed by an optional else part, which executes when the test condition is false.

Syntax:

```
if (test condition)
{
    statement;
    statement;
    ==========
    statement;
}
else
{
    statement;
    statement;
    ==========
    statement;
}
```

If the test condition is true, the if block statements will get executed otherwise else block statements will get executed

Pgm: Find square of a no. without header (math.h) ... input is 5.

```
#include<stdio.h>
int main()
{
    int n, s;
    printf("Enter any number:");
    scanf("%d", &n);
    s = n*n;
    if(n == 5)
    printf("\n the square number no. is %d", s);

    else
    printf("\n %d is not 5");

}
```

If else if stmt

If else-if stmt is executed in two forms as

1) If-else ladder
2) nested if else

(6)

If-else ladder

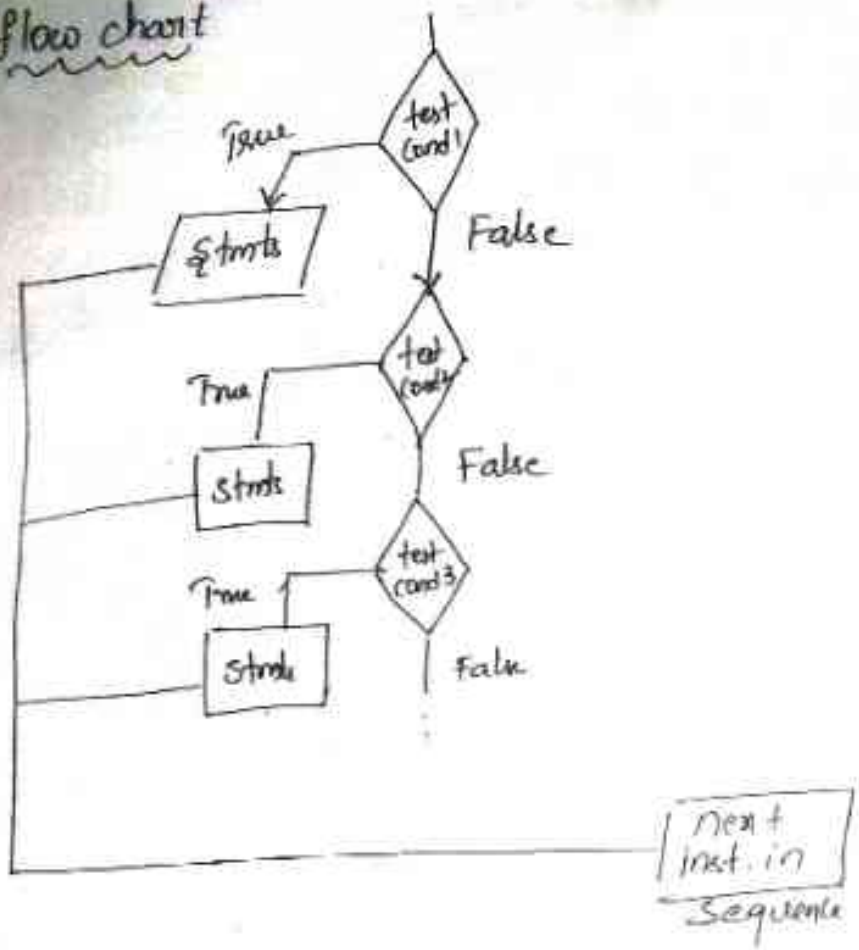
The if else ladder allows to choose b/w multiple test condition i.e. to select one of conditions in more than one test condition.

Syntax:

```
if(test condition)
{
    stmt;
}
else if (test condition)
{
    stmt;
}
else if (test condition)
{ stmt;
}

else if (test condition)
{ stmt;
}
else
{ stmt;
}
```

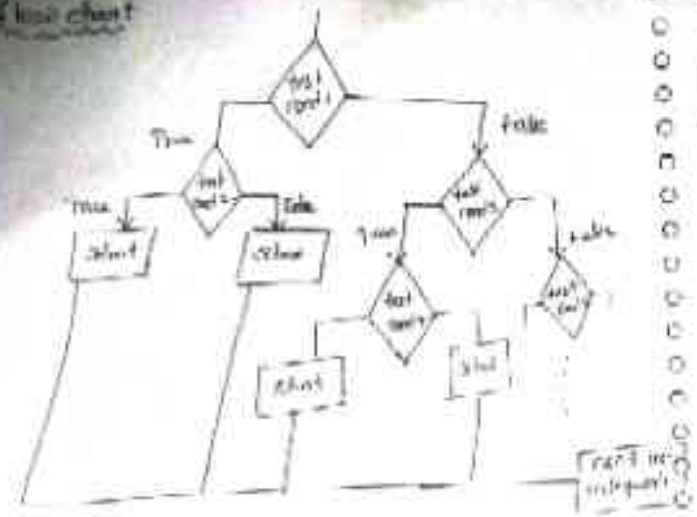
flow chart



Eg: relation b/w 2 Variables

```
#include<stdio.h>
int main()
{
    int a,b;
    printf("Input values for a & b:");
    scanf("%d%d",&a,&b);
    if(a>b)
    printf("\n %d > %d is the relation ",a,b);
    else if(a<b)
    printf("\n %d < %d is the relation", a,b);
    else
    printf("\n %d == %d is the relation", a,b);
}
```

flow chart



Eg:- max of two using nested if else

```
int main()
{  int a,b,c;
   printf("Enter values for a, b and c");
   scanf("%d%d%d", &a, &b, &c);
   if(a>b)
   {  if(a>c)
      printf("\n a is max among %d %d %d", a,b,c);
      else
      printf("\n c is max among %d %d %d", a,b,c);
   }
   else
   {  if(b>c)
      printf("\n b is max among %d %d %d", a,b,c);
      else
      printf("\n c is max among %d %d %d", a,b,c);
   }
}
```

switch - case

A switch 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.

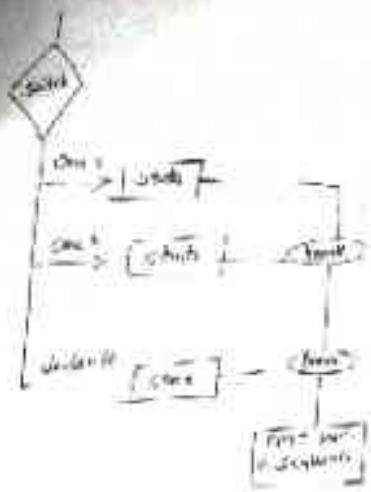
switch case tests the value of a variable and compares it with multiple cases. Once the case match is found, all statements associated with that particular case are executed.

If a case match is not found, then the default statement is executed and the control goes out of switch block.

Syntax :-

```
switch (variable)
{
   case constant-1: statement;
                    break;
   case constant-2: statement;
                    break;
   case constant-3: statement;
                    break;
   :
   case constant-n: statement;
                    break;
   default : statement;
}
```

Flowchart



Switch Calculator

```
int a, b, c; char ch;
printf("Enter two numbers and an operator: ");
scanf("%d %d %c", &a, &b, &ch);
switch(ch)
{
  case '+': c = a + b;
            printf("%d + %d = %d", a, b, c);
            break;
  case '-': c = a - b;
            printf("%d - %d = %d", a, b, c);
            break;
  case '*': c = a * b;
            printf("%d * %d = %d", a, b, c);
            break;
  default: printf("Invalid operator");
}
```

Advantages of switch

a) Easier to read than equivalent if-else ladder.
b) Faster in debugging
c) Easier to maintain.

Q. program to check even or odd using goto stmt.

```
#include<stdio.h>
int main( )
{
    int n;
    printf("Input value for n:");
    scanf("%d", &n);
    if (n%2==0)
    goto even;
    else
    goto odd;
    even:
        printf("\n %d is even", n);

    odd:
        printf("\n %d is odd", n);

}
```


Loop Stmts (Iteration Stmts)

A loop executes the sequence of stmts until the specified condition is true. A loop consists of two parts, a body of a loop and a control stmt. The purpose of the loop is to repeat the same code a number of times.

three types of loops in C programming are:

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

A looping process generally has following steps:

1) Setting and initialization of loop variable. Variable = 0;
2) test for a specified condition
3) execution of the stmts in the loop
4) increment/decrement of loop variable
5) ...

while loop

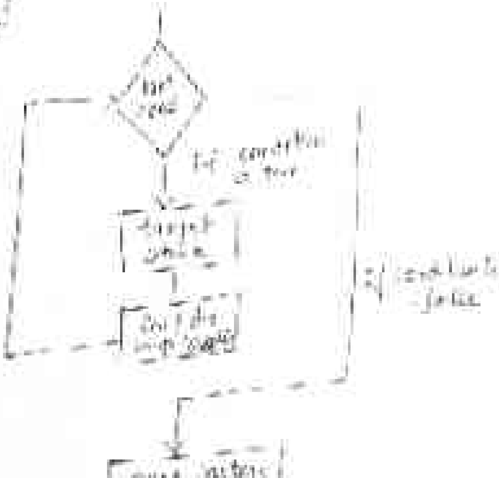
deciding how many times a certain action is to be done. A while loop in C repeatedly executes target stmt as long as a given condition is true/false.

Syntax :-

initialize loop counter var;
while (test condition)
{
statement 1;
statement 2;

Updating
Incr/dec loop counter var;
}

Flow chart



condition is checked before executing the body of the loop, in while loop.

→ It is also called as entry controlled loop (or) pre-checking loop/entry testing loop.

→ In while loop, if the condition is not true, then the body of a loop will not be executed, not even once.

Note :- The condition in the while loop should eventually become false at any point in the loop, it becomes to finite loop. (Endless loop) The loop that does not stop executing and processes the infinite number of times is called as infinite loop.

Sum of the series upto a given limit using while loop

```
int main()
{
    int n, i=1, sum=0;
    printf("Input the limit value : ");
    scanf("%d", &n);
    while (i<=n)
    {
        sum = sum+i;
        i++;
    }
    printf("The sum of the series upto given limit is %d", sum);
```

for the first time, the statement in the do-while loop get executed without any condition

flowchart

For loop

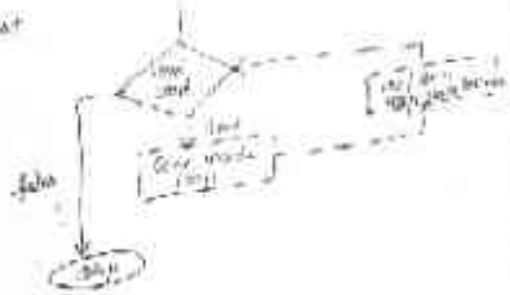
A for loop is a repetition structure which allows us to write a loop that needs to execute a specific number of times.

Syntax :

```
for(expression1; expression2; expression3)
{
    statement1;
    statement2;
    ---
    statementN;
}
```

where expression1 - initialization
expression2 - test condition
expression3 - increment or decrement

Flow Chart



Note :

Multiple initialization and step can be done in single for loop

Eg:

Arrays

An array is a collection of data items, all of the same type, accessed using a common name. Stored in contiguous memory locations.

Declaring an array in java is as,

Syntax

```
datatype arrayname[size];
```

Eg: int a[5];

Here int is the data type, a is the array name and 5 is the array size i.e. the array can hold 5 data items of integer type. Each of the individual data items of an array is called as elements of an array. Each element is accessed by its index. Index starts with 0 and ends with (size-1).

Eg:- int a[5];

→ 1st element of the array is indexed as 0 and accessed as a[0]

→ Likewise 2nd element is indexed as 1 and accessed as a[1]

→ 3rd element indexed as 2 and accessed as a[2]

→ 4th element indexed as 3 and accessed as a[3]

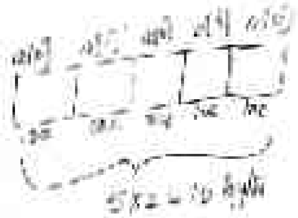
→ 5th element indexed as 4 and accessed as a[4]

So the 5 elements are a[0], a[1], a[2], a[3], a[4].

Memory Allocation

Immediately after declaring an array, compiler allocates that much of memory according to array size and type and points to the address of the first element called as base address of the array. By this reason we can also called array is known as [illegible] type of variable.

Eg:- int a[5];



(13)

Properties of Arrays

1) An array holds elements that have the same data type.

2) Array elements are stored in subsequent memory locations.

3) Array name represents the address of the starting element (base address).

4) Array size should be mentioned in the declaration, which is a non-zero positive number.

Initializing An array

Arrays can be initialized by any one of the following ways:

1) Initialization during array declaration

2) Initializing array in the middle of the program

Initialization during array declaration

Syntax:

datatype arrayname[size] = { list of initialization };

Eg:- int a[5] = { 1, 3, 5, 7, 9 };

a[0] = 1

a[1] = 3

a[2] = 5

a[3] = 7

a[4] = 9.

Note:-

* While initializing the array during declaration, if the no. of initialized are less than the no. of elements in the array, the remaining elements are assigned to zero.

Eg: int a[6] = {2, 4, 6, 8, 10};

a[0] = 2   a[2] = 6   a[4] = 10
a[1] = 4   a[3] = 8   a[5] = 0

* It is an error to specify more no. of initializers than the no. of elements in the array.

Eg: int a[3] = {1, 2, 3, 4, 5};
compile error.

Initializing array at the middle of the program

Initializing an array means some space in the program is allocated, each defined such that some constant value is initialized for the elements of an array.

Eg:
int marks[6];
for (i=0; i<6; i++)
marks[i] = 0;

Types of Arrays

Arrays can be classified according to the no. of subscripts associated with it as

1) One dimensional Arrays
2) Two dimensional Arrays
3) Multi dimensional Arrays

One Dimensional Array

An array which has only one subscript is known as one-dimensional / single dimensional / 1-D / linear array / single subscripted.

Syntax

datatype arrname[size];

Eg: int a[5];

↓ data type
↓ arr name
↓ size (no. of elements)

Operations performed on Arrays

The various operations supported by an array -

1) Traverse - print all the array elements one by one.
2) Insertion - add an element at given index.
3) Deletion - delete an element at given index.
4) Search - search an element using given index/value.
5) Sort - sorting all the elements of an array.

Traversal:-

Visiting each element once, also called visiting of an array.

Searching Techniques (Unit 3) (15)

Searching is an operation which helps to find the place of a given element or value in the array. Searching is said to be successful or unsuccessful depending upon the element which is being searched i.e. present or not in the array.

The commonly used searching techniques are

1) Linear search (or) Sequential search
2) Binary search

Linear Search (or) Sequential Search

This is the simplest method for searching. This method can be performed on a sorted or unsorted list (array). In this method each element of the list is sequentially visited and checked whether it is the desired item that is being searched for. To find the no. of comparisons for a successful search, we find the no. of comparisons for each element in the list. By the same way for all elements in the list.

1) Key matches with 1st element - 1 comparison
2) Key matches with 2nd element - 2 comparisons
3) Key matches with 3rd element - 3 comparisons
n) Key matches with nth element - n comparisons

So, the total no of comparisons = $\frac{1+2+3+\cdots+n}{n}$

$= n(n+1)/2n$

$= \theta(n^2)$ $n/2n$

$= O(n)$

Algorithm

step1: Read the Search element (key)

step2: Compare the search element (key) with all the elements of the list starting with the first element (a[0]) and [illegible]

step3: If there is a match, then display key element found and terminate from the loop.

step4: If not, repeat steps 2 and 3 until key element is compared with [illegible] element in the list.

step5: If last element in the list also [illegible] match with the key element, then display key element not found and [illegible] from the program.

Linear Search program

```
main( )
{
  int a[20], n, i, key;
  printf("Input array size:");
  scanf("%d", &n);
  printf("\n Input array element:");
  for(i=0; i<n; i++)
  scanf("%d", &a[i]);
  printf("\n Input the key element to be searched:");
  scanf("%d", &key);
```



```
  for (i=0; i<(n-1); i++)
  {
    if (a[i]==key)
    {
      printf("\n the key element %d is found
        at %d location", key, i+1);
      break;
    }
  }
  if (i==n)
  printf("\n key element not found");
}
```

Binary Search (half-interval search)

Binary search works only on a sorted list. So the list must be sorted to use binary search on a list.

Binary search follows divide and conquer approach. In Binary search technique, the list is divided into two halves and the key element is compared with the middle element of the list. If the match is found, then the location of the middle element is returned; otherwise, the search goes on in either of the halves depending upon the key element.

Result of the match -

a) If the middle element matches with the key element, the search is successful.

b) If the middle element is greater than the key element, the search [illegible] is in the [illegible]

...searched at in the first half.

c) If the middle element is less than the key element, then value which is searched is in the second half of the list.

Binary Search Algorithm

step 1: Start

step 2: Read an array consisting of n elements, key to be search.

step 3: Initialize low with position of first element and high with position of last element
low = 0
high = n-1

step 4: Check if low ≤ high value and calculate mid position,
mid = (low+high)/2

step 5: Check if a[mid] = key, if true return mid value and go to step 8

step 6: If a[mid] < key, then low = mid+1
high = mid-1

step 7: If a[mid] > key, then high = mid-1

step 8: Repeat same step 4 till step 7 to check if key exist in the list.

step 9: If end of the list is reached step 8 key element not found

step 10: Stop.

Program to implement binary search.

```
main()
{
  int a[20], n, i, key, low, high, mid;
  printf("Input array size:");
  scanf("%d", &n);
  printf("\n Enter array elements:");
  for(i=0; i<=(n-1); i++)
    scanf("%d", &a[i]);
  printf("\n Input key to be searched:");
  scanf("%d", &key); low=0, high=n-1
  while (low<=high)
  {
    mid = (low+high)/2;
    if (a[mid]==key)
    {
      printf("\n key element found is found at
        that position", key, mid);
      exit(0);
    }
    else if (a[mid]<key)
      low=mid+1;
    else
      high=mid-1;
  }
  printf("\n key element not found");
}
```

Bubble sort (sinking sort)

Bubble sort compares the adjacent elements and swaps them positions if they are not in order.

We start with the first element, compare the first and second elements, if they are not in order, they are swapped. Then we compare second and the third element, swap them if they are not in order and repeat the process until the last element.

Eg: 5 1 3 2 4

1 5 3 2 4

1 3 5 2 4

1 3 2 5 4

1 3 2 4 5 — P1

1 3 2 4 5

1 2 3 4 5 — P2

1 2 3 4 5

1 2 3 4 5 — P3

1 2 3 4 5

1 2 3 4 5 — P4

Note: In general if a list contains 'n' elements, then (n-1) passes (steps) are done in the bubble sort.

Program to implement bubble sort

```
#include<stdio.h>
int main()
{
    int a[20], n, i, j, t;
    printf("Enter array size\n");
    scanf("%d", &n);
    printf("Enter array elements:\n");
    for(i=0; i<n; i++)
        scanf("%d", &a[i]);
    printf("\n Entered array elements:\n");
    for(i=0; i<n; i++)
        printf("%d\n", a[i]);
    for(i=0; i<n-1; i++)
    {
        for(j=0; j<n-i-1; j++)
        {
            if(a[j]>a[j+1])
            {
                t=a[j];
                a[j]=a[j+1];
                a[j+1]=t;
            }
        }
    }
    printf("\n Array after sorting:\n");
    for(i=0; i<n; i++)
        printf("%d\n", a[i]);
}
```

2D Array | Matrix

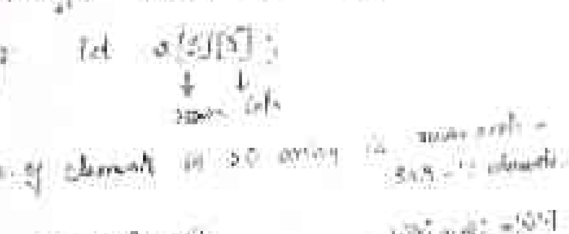
2D array can be represented as a table of rows and column represented as matrix.

Syntax

datatype arrayname[rowsize][colsize];

Eg: int a[5][5];
↓ ↓
rows cols

no. of element in 2D array is = rows × cols = 5×5 = 25 elements



In a matrix, the elements are stored first row i.e. after storing the elements of its first row the second row elements are stored and so on up to the last row.

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

a[0][0] = 1
a[0][1] = 2
a[1][0] = 3
a[1][1] = 4

Note: The array elements are accessed with row and col index.
i.e. a[1][0]
↓ ↓
Row value, Col value

(50)

Operations performed on 2D array:

i) Reading / displaying elements of a matrix (Standard)
ii) Addition / subtraction / multiplication of matrices
iii) transpose, symmetric
iv) [illegible], [illegible], upper, lower, identity, null matrix

Program to read / display elements of a matrix.

```
int main ()
{
    int a[5][5], r, c, i, j;
    printf("Input rows & cols");
    scanf("%d%d", &r, &c);
    printf("\n Enter array elements : ");
    for(i=0; i<=(r-1); i++)
    {   for (j=0; j<=(c-1); j++)
        {
            printf("\n a[%d][%d] = ", i, j);
            scanf("%d", &a[i][j]);
        }
    }
    printf("\n Array elements : \n");
    for (i=0; i<=(r-1); i++)
    {   for (j=0; j<=(c-1); j++)
        {   printf("\n a[%d][%d] = %d", i, j, a[i][j]);
}   }   }
```

Multi-dimensional arrays

A multi-dimensional array is an array of arrays i.e. an array with three or more dimensions. [illegible]

Syntax

datatype arrname[d1][d2][d3]...[dn];

d1 → 1st dimension
dn → size of final dimension.

Eg:- int a[2][2][2] = {1,2,3,4,5,6,7,8};

no. of arrays, no. of rows, no. of cols

2 arrays with 4 elements each, which can be represented as,

a[0][0][0] = 1	a[1][0][0] = 5
a[0][0][1] = 2	a[1][0][1] = 6
a[0][1][0] = 3	a[1][1][0] = 7
a[0][1][1] = 4	a[1][1][1] = 8

program to read & display multi dimensional array

```
#include <stdio.h>
int main()
{  int a[2][2][2], i, j, k;
   for (i=0; i<2; i++)
   {  for (j=0; j<2; j++)
      {
          for (k=0; k<2; k++)
          {
              printf("\n a[%d][%d][%d] = ", i, j, k);
              scanf("%d", &a[i][j][k]);
          }
      }
   }
   printf("\n the elements of the multi dimensional
           array \n");
   for (i=0; i<2; i++)
   {
      for (j=0; j<2; j++)
      {
         for (k=0; k<2; k++)
         {   printf("\n a[%d][%d][%d] = %d",
                    i, j, k, a[i][j][k]);
         }
      }
   }
}
```

Strings

Strings are actually one-dimensional array of characters terminated by a null character ('\0') i.e a string is defined as character array (or) an array consisting of characters terminated by null character.

The null character indicate the end of the string and used as a signal for the compiler ensure that the string has ended.

A string which is not terminated by a null character is not really a string but a collection of characters.

To represent a string, a set of characters are enclosed within double quotes (" "). when ever compiler encounter set of characters in double quotes it automatically inserts null character at the end of the string.

Declaring a string

Syntax:- char stringname[size];

Eg:- char college[10];

↓ name of the string

↓ no. of characters including null character

i.e college can store 10 characters including the null character (string terminated).

Initializing a string variable

We can initialize a string variable in two different ways as,

char college[10] = "NSCHUKSB";

(or)

char college[10] = {'N','S','C','H','U','K','S','B','\0'};

In the first one, compiler automatically insert null character ('\0') at the end of the string when it encounter set of characters in double quotes.

Where as in the second example, it is initialized character by character and we need to explicitly insert the null character to call it as a string.

Reading strings

There are 2 ways to read a string variable from input terminal, by using scanf( ) and gets( ).

scanf( )

We can read strings using scanf(), with the syntax as,

scanf("format specifier", &v1, &v2, ... &vn);

Eg:- char college[10];

scanf("%s", college);

We skip the address specification for a string variable because the string is declared as an array of characters (character array). The compiler, after declaring a string, can compute/allocate that many bytes of memory according to the size and it points to the address of the first character of the string called as base address. So we don't require address specification.

Similarly the string name is used without subscripts ([ ]) in the scanf(), which tells how many characters according to the size of the string, to define only one single string variable.

Note:- using scanf() we can't read a multi-word string.

Eg:- M's college → only M's is read; since space is a delimiter in scanf.

gets( ) :- we also can read a string variable using gets().

Syntax :- gets(stname);

Eg:- char college[10];

gets(college);

We can read a multi-word string using gets() until the size of the string.

String Handling functions :-

With every C compiler a large set of useful string library functions are provided and the header file <string.h> is used for all string library functions.

Function name	Meaning
strlen( )	Returns the length of the string (count of characters)
strlwr( )	Converts string in uppercase to lower case
strupr( )	Converts string in lower case to upper case.

strcpy( )	Copies one string into another
strcmp( )	Compares two given strings
strcat( )	Appends one string at the end of another string
strrev( )	Reverses the given string

strlen( )

this function returns the count of the characters present in string excluding the null character.

Syntax : identifier = strlen(strname);

strlwr( ) :-

converts the given string in uppercase to lower case.

Syntax : strlwr(strname);

strupr( ) :

Converts the given string in lower case to upper case.

Syntax :- strupr(strname);

strrev( ) :- this function is used to reverse the given string.

Syntax :- strrev(strname);

strcpy( ) :- Copies one string into another string

Syntax :- strcpy(strname, strname);

here string is copied into string

Eg:- strcpy(str, " manju");

Manju is copied into str.

strcmp( ) :- Compares two given strings and returns and of any of the strings.

Syntax :- strcmp(str1, str2);

→ if both the strings are equal strcmp returns zero or else 1/-1 according to the mismatch of ASCII value.

strcat( ) :- concatenates two given strings i.e appends second string at the end of first string.

Syntax :- strcat(str1, str2);

Array of strings :-

To represent a list of students or employees in an organization we use array of string.

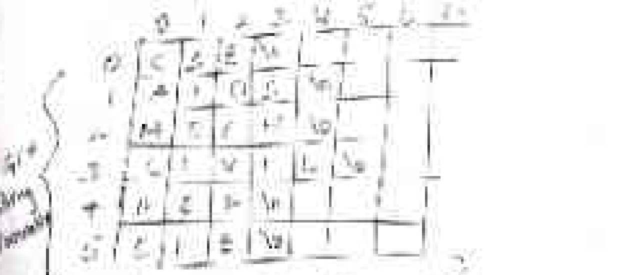
Syntax :-

char name[no of string][no of char];

Eg: char name[60][20];

↓ no. of string allocated → length of each string max.

char Branch[6][10] = { "[illegible]", "[illegible]", "[illegible]", "[illegible]", "[illegible]" };



Memory allocated = 6×10 = 60 byte.

Functions

A function is defined as a block of code which does a particular task.

C language has 2 types of function

i) built in / library functions :- these are already defined functions into library. The header file [illegible] which has their definition.

Eg: printf(), scanf(), gets(), puts [illegible]

(stdio.h header file)

ii) user defined functions :- defined by the user for a given task.

Advantages of functions

i) top down approach - divide a large / complex problem task to simpler versions.

ii) Avoids over writing of code in a program when of [illegible] lines are to be repeated in the program many times i.e. define a function once and call it when ever required in the program.

Syntax of user-defined function

function header

returntype functionname (list of arguments)

{

Local declarations;

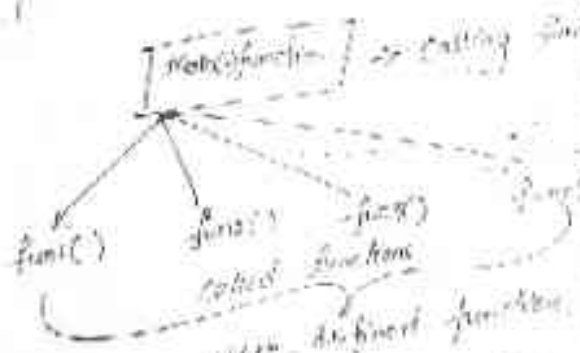
statements;

=

return expression;

}

body of user-defined function



*) main() is a special type of user defined function

*) main() is called as calling function since execution of any C program begins by executing main() and it is one mandatory function in all the C programs

*) the user defined functions which are called by the main are called as called functions.

*) It is from the calling part of the program - which is always main() the called functions get executed

*) the user defined functions called from - some main() execute in their own.

→ In the syntax of user defined functions the returntype indicates the type of value which is returned by the user defined function i.e. int, float, char, double etc.

→ this function name is used to identify the function by which we have many the function in the program.

→ The variables used in implementing the functions are called as arguments. To execute the function we require some arguments which are to be passed to the called function from calling function.

→ Some times apart from the values returned from calling function the called function has own local variables to execute the function.

→ After implementing the logic, the function has to return the computed value to the calling portion of the program which is main() using return stmt.

→ The return stmt primarily helps to return back to the calling function.

→ The arguments (variables) which are used to implement the function are of 2 types:
- actual arguments or actual parameters
- formal arguments & formal parameters

→ The actual parameters are always present in the calling portion which is always main()

→ The formal arguments are present in the called fn which is the user defined function.

Eg:- (2) Program to add 2 nos using functions.

```
#include<stdio.h>
int add (int, int);  -> function declaration & prototype
main()
{
  int a,b,s;
  printf("Input values for a & b");
  scanf("%d %d", &a, &b);
  s = add (a,b);   -> function call.
  printf("the sum of %d + %d = %d",
                a, b, s);
}
int add (int x, int y)  --> function definition
{
  int z;
  z = x+y;
  return z;
}
```

→ here a & b are actual arguments
→ when fun call the actual values from calling part - main() are passed to formal arguments.
→ x & y are formal arguments, they have ... actual.

→ the very first line after header file is known as function declaration or function prototype.

As function comes as an example for derived datatype, before using a variable it has to be declared, before defining the function we declare it which is known as function declaration or prototype.

In prototype the naming of the arguments is not compulsory we just tell the no. and type of arguments used in the function.

We write a semicolon (;) compulsorily at the end of function prototype or declaration.

→ As we know the execution of any C program begins by executing main() i.e. the calling function, the control from the calling part is transferred to the called part using function call without which we cannot execute the called function.

→ It is through function call the actual arguments from calling part are passed to called function.

Note :-

*) we cannot define a function inside another function.

*) function definition does not end with semicolon (;) as we are defining what it has to do.

*) the return statement can only return one value at a time from the called function.

*) we can have multiple return statement in a function but a function can return only one value when ever it is called.

*) we can define function i.e. called function before or after calling function but the execution always begins by executing main() in C language.

*) we can have more than one user defined function in a single program.

-> after implementing the function, there comes which is to return part of the program is returned back to the calling part which is with help of return stmt

program to find factorial using function

```
#include<stdio.h>
int fact(int); -> declaration/prototype
main()
{
    int n, f;
    printf("Input a number:");
    scanf("%d", &n);
    f = fact(n); -> function call, n is actual arg
    printf("\n Factorial of given no %d = %d", n, f);
}
int fact(int n) -> fun definition
{                  n -> formal arg.
    int i, f=1;
    for(i=1; i<=n; i++)
      f = f*i;
    return(f);
}
```

Function Prototype / Function declaration

In c language, the default return type of any function is int.

If the function returns a value other than integer, we need to declare the function, before invoking it, which is known as function declaration or function prototype.

Syntax :-

returntype functionname(datatype arg1, datatype arg2, ... datatype argn);

Here, the returntype indicates the type of value returned by the function.

Note :-

a) A semi colon (;) is used in function prototype i.e. it ends with semi colon.

b) Naming of arguments is not mandatory in function prototype.

Eg:- int fact(int);

Function prototype

- Function definition

*) It ends with a semi colon (;)

*) Naming of arguments is not compulsory

*) Very first line in the program even before main()

*) Syntax :-

return-type function name (list of args);

No semicolon at the end

Naming of arguments is must

Must be declared before or while defining main()

Syntax :-

return-type function name (list of args)

{

local declarations;

(if any)

statements;

return expression;

}

Note :- If the function doesn't return any value to the calling function the return type of the function should be specified as void — data type specifier

Types of user defined functions

According to the presence of argument and return value, the defined functions can be classified as,

*) Function with no return & no arguments

*) Function with no return & arguments

*) Functions with return & no arguments

*) Function with arguments & return

Function with no return & no arguments

The calling function does not get any value from the called function like wise the called function doesn't return any value to the calling function

```
void function1();
main()
{
  function1();  => no arguments
}
void function1()
{
  ... no return stmt
}
```

Functions with no return & with arguments

The called function receives arguments from the calling function but the called function should not return any value to the calling function.

Eg:- void funname (int, arg);

main( )

{

some value to pass to fun

funname (arg);

}

void funname (list of args)

{

no return stmt.

}

Functions with return and no arguments

The calling function doesn't pass any value to the called function but the called function returns value to the calling function.

Eg: returntype funname( );

main( )

{

funname( ); --> no arguments

}

returntype funname( )

{

return expression; --> return stmt in called fun.

}

Functions with return and with arguments

Called function receives arguments from the calling function and with return the value to the calling fun.

Eg: returntype funname (arguments);

main( )

{

funname (args); --> with arguments

}

returntype funname (arguments)

{

return expression; --> return stmt

}

Passing arrays as an argument to function

We can pass arrays of any dimension as an argument to fun, in two ways:

a) Passing a single element of an array
b) passing entire array elements, i.e all the elements of the array

1) Passing a single element of an array

We can pass single element of an array as an argument to function using its index value.

Eg:-
```
void display(int);
main( )
{   int a[5]= { 1,3,5,7,9 };
    display(a[2]);
}
void display(int x)
{   printf("\n the element received is :
             %d", x);
}
```

the element indexed as a[2] i.e 5 is passed to display( ), it displays 5.

Passing all the elements of the array to a function

We can pass all the elements of an array variable as an argument to function simply by using array name

Eg:-
```
void display(int [ ], int);
main( )
{
    int a[5]= { 1,3,5,7,9 };
    display(a, 5); → passing array name & no. of elements
}
void display(int a[5], int n)
{
    int i;
    for(i=0; i<=(n-1); i++)
    printf("\n a[%d] = %d", i, a[i]);
}
```

here when we pass array name as an argument to a fun, the meaning is that we are passing the base address of the array i.e the address of the array name, equal contains the address of the first element and it is sufficient to access all the elements of the array

display(a, 5)

Parameter Passing Techniques

Whenever the control is transferred from calling fun. to the called fun. (on function call), the called function receives arguments from calling function i.e. the actual value gets binded with the formal value (positional parameters).

We have two techniques to pass arguments to called function from calling function as

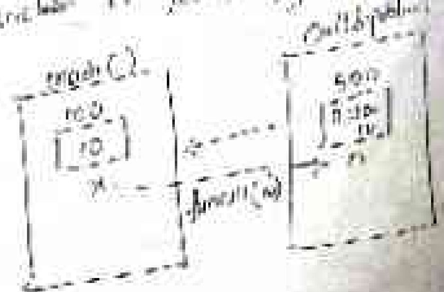
i) Call by Value / Pass by value
ii) Call by reference / Pass by reference

Call by Value

In this technique it is the actual value which is passed to the formal value from fun. call. We only pass value, not any thing else.

i) In this technique the changes made to the actual value are not reflected back to the calling function.
ii) At a time only value can be returned from called function to the calling function.

```
#include<stdio.h>
void callbyvalue(int);
void main()
{
    int x=10;
    printf("\n the value of x before calling the function is %d", x); → 10
    callbyvalue(x); → fun call
    printf("\n the value of x after calling the function is %d", x); → 10
}
void callbyvalue(int x)
{
    x=x+10;
    printf("\n the value of x in the function is %d", x); → 20
}
```



Inter-function Communication

While executing programs which use functions, the control is transferred from calling function to called function and after execution of called function, finally comes back to the calling function. Both calling and called functions have to communicate with each other to exchange information which is known as inter-function communication. In C language, inter-function communication can be done in 2 ways as,

1) Command communication
2) Upward communication
3) Bi-directional communication

Command Communication

- The data is transferred from calling function to called function but not from called function to calling function.
- Functions without parameters and with return value

```
return value
void fun();
main()
{
  fun();
}
void fun()
{
  ==
}
```

(command communication)

Upward Communication

- The data is transferred from called function to calling function but not from calling function to called function.
- Functions without parameters and with return value

```
returntype funname();
main()
{
  funname();
}
returntype funname()
{
  ==
  return expression;
}
```

(upward communication)

Bi-directional Communication

- The data is transferred from calling the called function and from called to calling function.
- Functions with parameters and with return value.

```
returntype funname(args);
main()
{
  ==
  funname(args);
  ==
}
returntype funname(args)
{
  ==
  return expression;
}
```

(bi-directional communication)

Recursion, Structures and Unions - IV unit.

Recursion is a process by which a function calls itself repeatedly until some base condition has been reached.

A function is called as a recursive function if a stmt within the body of the fun. calls the same function.

Recursion is a process of defining something in terms of itself.

When a recursive fun. is executed the recursive fun. calls are placed on a data structure called as stack until some base condition is reached as the recursive calls are not executed immediately.

Eg:- $n! = n*(n-1)!$

$= n*(n-1)*(n-2)!$

etc.

if suppose $n=5$

$5! = 5*4!$

$= 5*4*3!$

$= 5*4*3*2!$

$= 5*4*3*2*1!$

$= 5*4*3*2*1*0!$

$= 5*4*3*2*1*1$

Last five answers

repeats until a base condition is reached i.e factorial of zero is one. ↳ base cond.

Non-recursive fun

```
for(i=0; i<=n; i++)
f = fact
```

same memory gets updated if n=5

Factorial using recursion

```
#include<stdio.h>
int fact(int); → prototype
main()
{
    int n, f;
    printf("Input a value:");
    scanf("%d", &n);
    f = fact(n); → fun. call
    printf("\n the factorial of %d is %d", n, f);
}
int fact(int n) → fun. def
{
    if (n==0)
    return 1;
    else
    return (n * fact(n-1));
          → function recursive fun.
}
```

Tracing if n=5

stack

fact(5)

n==0 false

else

5 * fact(4)

[illegible]

Assignment to find base raised to exponent

```
#include<stdio.h>
int power(int, int); → prototype
main()
{
    int b, e, p;
    printf("Input value for base & exp");
    scanf("%d %d", &b, &e);
    p = power(b, e); → fun. call
    printf("\n the value of the base %d raised to exponent %d is %d", b, e, p);
}
int power(int x, int n) → fun. def
{
    if (n==0)
    return 1;
    else
    return (x * power(x, n-1));
}
```

if b=2, e=3

power(2, 3) x=2, n=3

return 2 * power(2, 2)

2 * 2 * 1

2 * 2 * power(2, 1)

2 * 2 * power(2, 0)

2 * power(2, 2)

Tracing stack

GCD of 2 nos

```
#include <stdio.h>
int gcd(int, int);  → prototype
main()
{
    int a, b, g;
    printf("Input value for a and b:");
    scanf("%d%d", &a, &b);
    g = gcd(a, b);  --- fun call
    printf("\n GCD of given nos %d and
            %d is %d", a, b, g);
}
int gcd(int x, int y)  --- fun def
{
    if (y==0)
    return x;
    else
    return (gcd(y, x%y));
}
```

Eg a=4, b=6
x=4, y=6

gcd(2, 0)
gcd(2, [illegible])
gcd(6, 4%6)

Binary Search using recursion

a[5] = {1, 3, 5, 7, 9};

1	3	5	7	9
0	1	2	3	4

key = 3, l=0, h = length-1

Logic: In the binary fun. we pass array elements, no. of elements, key value, low and high values etc.

```
binary(int a[], int n, int key, int l, int h)
{   m = (l+h)/2;  → calculate mid;
    if (l > h)
    return -1;
    else if
    (a[m] == key)
    return m;
    else if (a[m] > key)
    binary(a, n, key, l, m-1);
    else
    binary(a, n, key, m+1, h);
}
```

function call to the same fun

binary(a, 5, 3, 0, 4)
m = (0+4)/2 = 2

a[2] == 3 ✗
binary(a, 5, 3, 0, 1)
m = (0+1)/2 = 0
a[1] == 3 ✓
return (1) - the position of key element

binary(a, 5, 3, 0, 4)
binary(a, 5, 3, 0, 1)

Points to remember while using recursion (1)

1) When ever we use recursion to execute a program, no loops should be used.
2) As we don't have immediate answers in recursive programs, the recursive calls are placed on separate memory blocks called as stack.
3) Stack is the data structure which is used to implement recursive programs.

Advantages of Recursion
1) Sometimes easy to solve using recursion than iterative solution, which is compiler [illegible]
2) Unnecessary [illegible]

Disadvantages
1) Solution is always logical & [illegible]
2) Takes lot of space
3) Uses more processing time

Note: If we don't converge to base condition in recursion, we end up with infinite loop. At some time the base condition must be true at some time, whereas in loops, the test condition should be false, otherwise we end up with infinite loop.

Structures (2)

A structure is a collection of one or more variables of different datatypes grouped together under a single name for convenient handling.

Syntax:

```
struct <struct name/tag>
{
    datatype member1;
    datatype member2;
    datatype member3;
    ----
    datatype membern;
}structurevariables;
```

Eg:

```
struct student
{
    int rno;
    char name[20];
    float avg;
}s1,s2,s3;
```

here the name of the structure as tag of the which is student defines a new datatype called as derived datatype

we have three variables s1,s2 & s3 which are of student type.

The structure definition does not reserve any memory space. It just gives a format of a new user defined datatype.

→ The memory is allocated for the structure variables s1, s2 and s3 which contain three fields as rno, name and attend as its member.

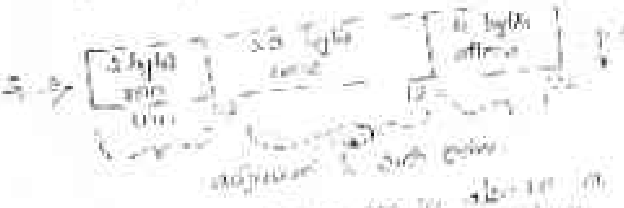
→ The memory allocated for structure variable is the memory allocated for all of its members.

Eg:- For structure variable s1, the memory is created as 2bytes 20bytes 4bytes

↓ rno ↓ name ↓ attend

Similarly 26 bytes are allocated for s2 & s3 also.

→ The memory allocated for a structure variable is always in sequential location.



adjacent to each other

The same case for s2 and s3 also i.e. in case of 26 bytes is allocated where the members are stored adjacent to each other

Note:- A structure is always defined but not declared.

Accessing Structure Members

The members of the structure can only accessed using variable of a structure i.e only then structure variables can be access the members of the structure using member access operator (or) member operator (.) dot operator with the syntax.

structurename.membername;

Eg:- struct student
{
int rno;
char name[20];
float attend;
}s1,s2,s3;

s1 → scanf("%d%s%f", &s1.rno, s1.name, &s1.attend);

s2 → scanf("%d%s%f", &s2.rno, s2.name, &s2.attend);

likewise even for s3.

Initializing a Structure Variable

We can initialize structure variables like we initialize the array variable.

Syntax:- struct structure name structure variable = {list of values};

Eg:- struct student s1 = {1, "[illegible]", 90};
struct student s2 = {2, "Archana", [illegible]};
struct student s3 = {3, "[illegible]", [illegible]};

Valid operations on structures

1) Reading and displaying values to members of structure using dot operator.
2) Knowing the size of the structure using sizeof( ) operator.

Eg: sizeof(s1); → returns size of structure variable s1 [illegible]

3) Assigning one structure variable to another using assignment operator.

Eg: s2 = s1; → s1, s2 must be of same type.

We can represent structures in any of the following 4 ways according to the requirement as,

a) Structure within structure (or) Nested structure
b) Array of structures
c) Arrays within structures

Structure within structure

A structure containing one or more structures as its members of structure is known as structure within structure or nested structure.

Eg:

```
struct employee
{
    int id;
    char name[20];
    struct address   → nested structure
    {
        char street[20];
        char city[20];
    } a1, a2;        → nested structure vars
    float sal;
} e1, e2;            → main structure vars
```

We have four members in the structure employee as,
- id — int
- name — string
- address — structure
- sal — float

The members of the nested structure are accessed with the system main structure.nested structure.member;

Eg:- e1.a1.street, e1.a1.city → for e1
e2.a2.street, e2.a2.city → for e2

Array of structure

C language permits the use of an group of data of different types stored in consecutive memory locations with a common variable name instead of [illegible] n different structure variables for the same structure. We can declare only one structure variable which is an array variable which is known as array of structures.

Eg:
```
struct student
{
    int rno;
    char name[20];
    float avg;
} s[120];
```

here s is a structure variable which is of student type and it holds data for 120 students as s[0], s[1], ... s[119].

The memory allocation is as:

Arrays within structure

We can use any dimensional array (i.e. one (1D) and multidimensional) of type int or float as members of structure.

Eg:
```
struct stud
{
    int rno;
    char name[20];
    int marks[5];   → int array as member of structure
} s[120];
```

Here to read marks of any student in 5 subjects we require an index s[i].marks[j], marks is an array in which marks[0], marks[1] and marks[4].

So to read these array we require a separate loop counter variable apart from the loop counter var which is used to represent student.

Eg: s[20].marks[1];
↓
20th student marks in 2nd subject

```
for (i=0; i<120; i++)
{ for (j=0; j<5; j++)
  { scanf("%d", &s[i].marks[j]);
```

Structures & Functions

Structures can be passed to functions as arguments in three methods as

(1) Each member of the structure is passed to the function

(2) Address of each member is passed to function

(3) Passing structure variable as an argument to function

Passing each member of the structure

the individual members of the structure can passed as actual arguments

as argument using call by value method

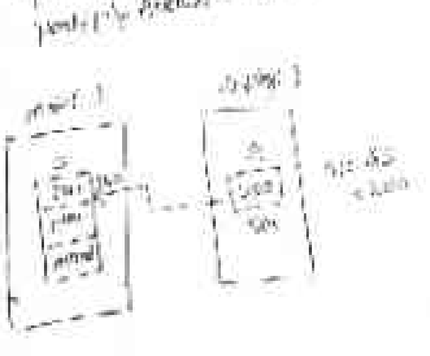
```
struct student
{ int roll;
  char name[20];
  float attend;
} s1, s2, s3;
void display(int, char [], float);
main()
{ display(s1.roll, s1.name, s1.attend);
  display(s2.roll, s2.name, s2.attend);
  display(s3.roll, s3.name, s3.attend);
}
void display(int r, char n[], float a)
{ printf("\n Roll No : %d", r);
  printf("\n Name : %s", n);
  printf("\n Attendance : %f", a);
}
```

(2) Passing entire structure using structure variable as an argument to a function using call by reference mechanism

```
struct student
{ int roll;
  char name[20];
  float attend;
} s1 = {1, "Manju", 98.5};
void display(struct student);
main()
{
  display(s1);
}
void display(struct student s)
{ printf("\n Roll No : %d", s.roll);
  printf("\n Name : %s", s.name);
  printf("\n Attendance : %f", s.attend);
}
```

(3) Passing address of a structure variable as an argument to a function using call by reference mechanism

```
#include <stdio.h>
struct student
{
    int rno;
    char name[20];
    float attend;
}s = {1, "manjiri", 90};
void display(struct student s1);
main()
{
    display(s1);
}
void display(struct student s1)
{
    printf("Roll no = %d", s1.rno);
    printf("Name = %s", s1.name);
    printf("Attendance = %f", s1.attend);
}
```



Unions

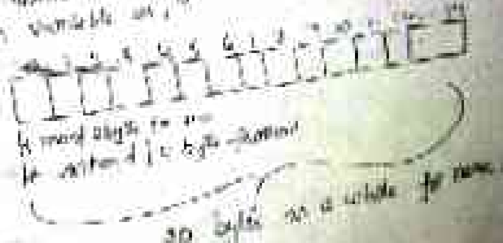
Union is a concept derived from structure therefore union follows the same syntax as that of the structure while defining.

Syntax:

```
union unionname
{
    datatype member1;
    datatype member2;
    datatype member3;
    ==
    datatype memberN;
}unionvar;
```

```
union student
{
    int rno;
    char name[20];
    float attend;
}s;
```

Memory allocation of a union is common to the memory of the largest member i.e. only 20 byte are allocated for a union variable s.



→ A major distinction b/w union and structure is in terms of memory allocation.

→ In structure each member has its own memory location whereas in union all the members share the same memory location.

→ the no. of bytes allocated for a union is the number of bytes required by the largest member in the union.

→ Since, all the members of the union share the same memory location, only one member of a union can be accessed at a time and hence we can't initialize all the members of union as they don't have individual memory.

Ex:

```
#include <stdio.h>
main()
{
    union student
    {
        int rno;
        char name[10];
        float attend;
    }s;
    s.rno = 1;
    printf("\n Roll no: %d", s.rno);
    s.name = "Manju";
    printf("\n Name : %s", s.name);
    s.attend = 98;
    printf("\n Attendance : %f", s.attend);
}
```

Bit fields

We can specify the size of members of structure or union in C language in bits i.e to use memory efficiently when we know the value of field will never exceed a limit or field with a small range.

Ex:

```
#include <stdio.h>
struct date
{
    unsigned int d: 5;
    unsigned int m: 4;
    unsigned int y: 12;
} dt = { 27, 11, 2020};

main()
{
    printf("\n size of the structure var is %d", sizeof(dt));
    printf("\n Date : %d/%d/%d",
           dt.d, dt.m, dt.y);
}
```

the size of the structure is 5 bits + 4 bits + 12 bits = 21 bits or 3 bytes.

Syntax:

struct <structname/tag>
{
datatype member1 : width in bits
unsigned int only.
——
}

Or the correct syntax is,

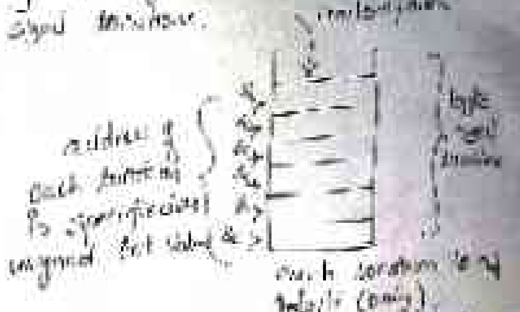
struct <structname/tag>
{
unsigned int member1 : width in bits;
unsigned int member2 : width in bits;
unsigned int member3 : width in bits;
——
unsigned int memberN : width in bits;
} structvar;

Note:-
1) We cannot have address of a bit field variable & we cannot use scanf() to read values into bit fields.

Pointers & Files - V unit

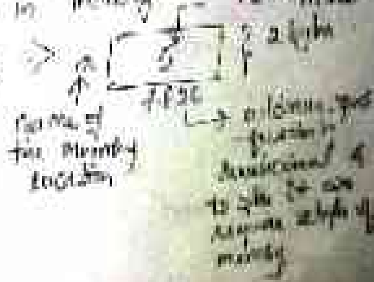
Memory Organization

→ Memory is organized as a sequence of byte sized locations. (contiguous)



→ The no. associated with each byte sized location to access the content stored in it is known as address which is a unique +ve unsigned integer value labeling a byte having fixed no of bits sized location in memory.

Eg: int a=5; =>



Because float h = 10.25;

h ← name of memory location

4 bytes { [10.25] ← value / content in memory location named as h.

8410 ← address of memory location labeled as h

here to store the address of memory location labeled as h we require 4 bytes.

Similarly char ch = 'A';

ch ← memory location name

1 byte { [A] ← value stored in memory location ch

8801 ← address of memory location named as ch

here also to store the address of memory location named as ch we require 2 bytes.

Note: irrespective of any type of value which is stored in a memory location, the address of memory location is specified by using 2 bytes.

Eg: to represent address of a student's roll number of branch and year are say 12 [illegible] or digits which is fixed — 1009 17 87001

like wise what ever might be the type of content stored in a memory location — in address specification always character is to be 2 bytes.

2) Every object which is stored in memory (instruction and data) is associated with a valid range of address.

Eg:

Addressing techniques

In order to access the contents of one or more memory location, we use addressing techniques which are of 2 types as,

1) Direct / Absolute addressing technique.

2) Relative Addressing technique.

Direct / Absolute Addressing technique

int a = 50;

a ← name / label of memory location [illegible]

2 bytes { [50]

1028 ← address of memory location labeled as a

here the address of a word is supplied to execute any instruction in the program which utilizes a. it is because of this reason this technique is called as direct or absolute addressing technique.

*) Relative addressing technique

In this technique address specification consists of a component of base address and offset address.

Eg: int a[5];

to access an element which is indexed as a[3] in relative addressing.

a[3] - base address + offset

= 100 + 3*2

= 106 => the value stored in location 106 is 3 which is value of a[3]

a[1] - base address + offset

= 100 + 1*2

= 102 => value stored in 102 location is 1 which is value of a[1]

Pointer Variables

A pointer is a variable which contains the memory address of another variable. i.e. pointer variable contains memory location rather than values.

If one variable contains the address of another, the first variable is said to point another variable.

The general syntax for declaring a pointer variable is,

datatype *ptrvariablename;

eg: int *ptr;
float *ptr;
char *ptr;

The above declaration informs the compiler:

a) allocate memory for ptr variable 2 byte for any type of pointer var.

b) associate the type of the pointer variable to the memory location.

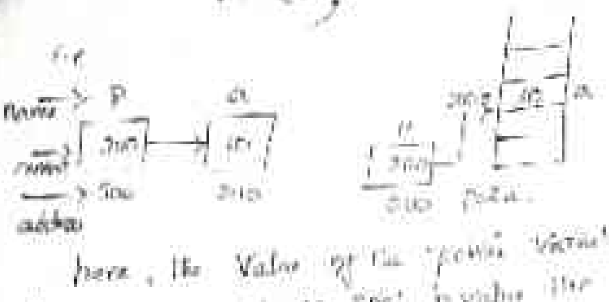
c) the pointer variable always contains the address of a variable which is of its type. i.e a ptr is an integer pointer variable, it can only store address of an integer variable and p is known as a pointer.

Initialization of a Pointer Variable

- The process of assigning address of a variable to a pointer which is of the same type is known as pointer initialization.

Eg:- int a = 10, *p;

p = &a;



here, the value of the pointer variable is 200 i.e. if we don't provide the address will be zero & the ... to point address of a variable i.e. 200.

In case if we observe, indirectly we are accessing the content stored in memory location with address 200 using pointer variable 'p', & to print the value stored at memory location pointed by p we use *p notation.

1) So if we say 'p' → it is the memory location value → 200.

2) And if we want to access the content stored in memory location 200 then p → it is *p.

Because of this reason, the operator * is known as indirection (or) dereferencing operator since indirectly we are accessing the content stored in a memory location using pointer variable 'p'.

3) The address operator (&) is known as reference operator.

Note :- Immediately after declaring a pointer variable it has to be initialized with address of some variable which is of the type.

=> If we are not initializing a pointer variable, it may lead to system crash i.e. a pointer variable can only store the address in its memory location. So if we are not initializing it properly it goes and holds an unpredictable value called as garbage value which leads to crash. So to avoid this we must initialize pointer variable immediately after declaring it.

Size of pointer variable

whatever may be the data type of pointer variable, the size of pointer variable is 2 bytes only i.e. we can declare a pointer variable of any type but the address specification of any memory location is an unsigned positive integer value which occupies (or) requires 2 bytes

Eg:-

```
int *a;   --> sizeof(a) --> 2 bytes
float *b; --> sizeof(b) --> 2 bytes
char *ch; --> sizeof(ch) --> 2 bytes
double *d; => sizeof(d) --> 2 bytes
```

Pointer Arithmetic

A limited set of arithmetic operations can be performed on pointers.

*) an integer may be added or subtracted to the value pointed by the pointer using operators like +, -, (or) - - -

Eg :-

```
int a=10, *p;
p=&a;
*p = *p+10; (or) *p+=10;
*p = *p-5; (or) *p-=5;
```


*) a pointer may be incremented or decremented using operators like ++ (or) -- in terms of its size.

Eg:-

```
int a=10, *p;
p=&a;
```

if p is incremented/decremented it moves forward or backward from where it is pointing to by 2 bytes.

i.e. p++ means 1500 & p-- means 1496

Invalid operations on pointers are

*) addition of two pointers.
*) multiplication of two pointers.
*) division of two pointers.
*) Mod of two pointers.

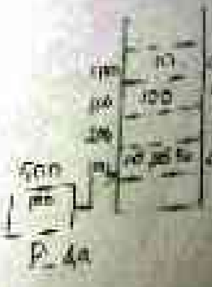
} to perform on these we require values but pointers can only store addresses.

Pointers can be used in different expressions as

i) Arithmetic expressions

Eg:-

```
int a=10, *p, b=2, c,d;
p=&a,               *p=10
*p+=10;
c=*p*5
 = 20*5 = 100
d= *p/b;
 = 20/2 = 10
```



P=&a

Note :- we can also modify the value of a variable thru its pointer as,

*p1=50 which is same as a=50

Subtracting one pointer from another

If suppose p1 contains the address as 208 and p2 contains the address as 200, then the stmt,

x = p1 - p2

will assign the no of array elements between p1 to p2 to x which automatically gets divided by the size of the datatype.

Eg: int a[5] = {1, 3, 5, 9, 7};

int *p1, *p2, x;

p1 = &a[4] = 208

p2 = &a[0] = 200

x = p1 - p2

= 208 - 200 = 8/2 (size of int is 2)

= 4

so we have four elements from one to another
a[3], a[2], a[1], a[0].

(ii) Assignment operation

A pointer can be assigned to another pointer if both are of same type.

Eg: int a=10, *p1, *p2;

p1 = &a;

p2 = p1;

*p1 = 20

*p2 = 30.

→ Value of p1 = 1000

→ Value stored at location pointed by p1.

(iii) Comparison operation

Pointers can be compared using equality (==, !=) and relational operator (<, >, <=, >=) operator (==) and

Eg: if (p1==NULL) → predefined macro whose value is zero, defined in all header files.

{
...
}

else

{
...
}

if (p1==NULL)
{
...
}
else
{
...
}

NULL pointer :- A NULL pointer is a pointer that does not point to any memory location. NULL is a built in constant whose value is zero which is similar to the NULL character in string.

If we don't have any valid address to be assigned to a pointer, we simply use NULL

Eg:

```
#include <stdio.h>
main( )
{
    int *p = NULL;
    printf("value of the var p is %u", p);
}
```

Arrays & Pointers

An array is always a constant pointer to its first element irrespective of any dimension in C language. At runtime/ accordingly after declaring an array according to the length and type of the array compiler allocates a block of memory and points to the address of first element called as base address.

int a[5];

&a[0] → base address

We can access array elements in one of the following usage

*) using subscript value
*) pointer offset notation using pointer name
*) pointer offset notation using array name.

using subscript value

Eg:
int a[5] = { 2, 4, 6, 8, 10 };
a[0]=2, a[1]=4, a[2]=6, a[3]=8, a[4]=10

We can access the elements of the array using subscript value with the syntax

arrayname[index];

i.e. like a[0], a[1], a[2], ... a[n-1];

pointer offset notation using pointer name

Eg:
int a[5] = { 2, 4, 6, 8, 10 }
int *p;
p = &a[0];

The elements of the array are accessed thru pointer var p with the syntax : *(p+i)

Eg: a[0]
= *(p+0)
= *(2000+0×2)
= *(2000)
= 2

a[1]
= *(p+1)
= *(2000+1×2)
= *(2002)
= 4

a[2]
= *(p+2)
= *(2000+2×2)
= *(2004)
= 6

a[3]
= *(p+3)
= *(2000+3×2)
= *(2006)
= 8

Pointer to access elements of 1D array.

```
#include <stdio.h>
main()
{
    int a[20], n, i, *p;
    p = &a[0];
    printf("Input the size of the array:");
    scanf("%d", &n);
    printf("Enter the array elements:\n");
    for(i=0; i<n; i++)
    {
        printf("\n a[%d]: ", i);
        scanf("%d", p+i);
    }
    printf("\n The elements of the array are:\n");
    for(i=0; i<n; i++)
        printf("%d\t", *(p+i));
}
```

Max & Min using pointer

2D Array & Pointer

Pointers & strings

Eg: char s[6] = "Hello";

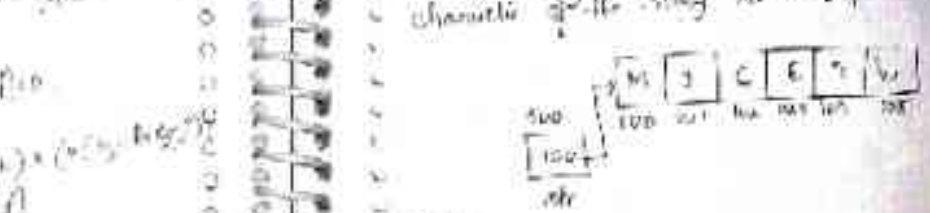
Compiler automatically inserts NULL character at the end of the string.

We can create strings using pointer variables of type char.

Eg: char *ch = "Hello";

The above statement creates a string and stores the address of the first character in the pointer variable ch.

The string variable ch points to the first character of the string Hello.



Note:

1) A string is defined in terms of memory not explicit value initialization with NULL is not required as, str = A; ✗

2) We can't read (or) scan the value of string pointer as it has to be initialized.

program using pointers to find the length of string

```
#include <stdio.h>
main( )
{
    int len;
    char *s;
    char *p = s;
    s = "MYCOLLEGE";
    printf("\n the string initialized is %s", s);
    while (*p != '\0')
    {
        printf("\n %c is stored at address location %u", *p, p);
        p++;
    }
    len = p - s;
    printf("\n the length of the given string is %d", len);
}
```

Eg:- initially s is in first character address
s = 500
even pointer to string p is initialized with s so given the address of p is also 500 which gets ac
until *p != '\0' i.e p which gets by calculating ... when the cond. is false ... so p value is 507
len = 507 - 500 = 7

Array of pointers

An array of pointers is an array variable which contains memory addresses as its values. i.e it is an array variable which is a collection of memory addresses or array of addresses.

Syntax :-

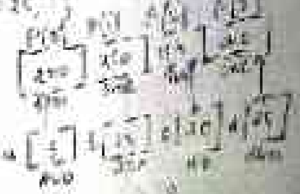
datatype *ptrname[length];

eg:- int *a[5];

→ here a is an int array which can store address of five elements of int type.

Eg:- program to illustrate array of pointers to access variables.

```
#include <stdio.h>
main( )
{
    int a=5, b=15, c=25, d=35;
    int *p[4];
    p[0]=&a;
    p[1]=&b;
    p[2]=&c;
    p[3]=&d;
    printf("\n the value of the variables are");
    for(i=0; i<=3; i++)
    printf(" %d\t", *p[i]);
}
```



Program to read different arrays of different size using array of pointers.

```
#include <stdio.h>
main( )
{
   int a[3]={1,2,3}, b[4]={2,4,6,8}, c[2]={[illegible]};
   int *p[3], i;
   p[0]=&a[0];
   p[1]=&b[0];
   p[2]=&c[0];
   printf("\n Elements of first array:\n");
   for(i=0; i<3; i++)
   printf("%d\t", *(p[0]+i));
   printf("\n Elements of second array:\n");
   for(i=0; i<4; i++)
   printf("%d\t", *(p[1]+i));
   printf("\n Elements of third array:\n");
   for(i=0; i<2; i++)
   printf("%d\t", *(p[2]+i));
}
```

Note:-

One important use of array of pointers is in handling a table of string i.e array of strings.

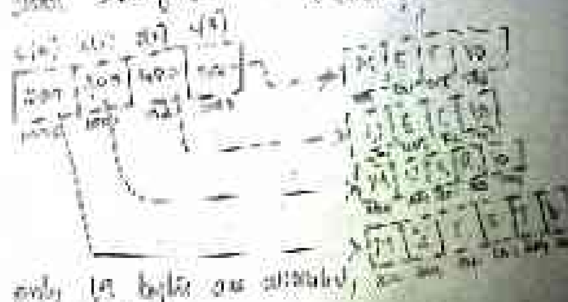
Eg:- char college[4][10]={"MSCET", "MVSR", "MJCET", "MTC"};

The memory allocation for it.

The above declaration tells that college is a table of string consisting of 4 names, each with a max length of 10 characters including null character.

So the total storage requirement is [illegible] length. In [illegible] memory, all the strings [illegible] of same length. Instead of making each row a fixed number of characters we make it a pointer to a string of varying length.

char *college[4]={"MSCET", "MVSR", "MJCET", "MTC"};



Here only [illegible] bytes are allocated, instead of [illegible].

Pointer to function

It is possible to declare a pointer pointing to a function, which can be used as an argument in another function.

Syntax :-

```
datatype (*pointername)(arg type);
```

Eg:

```
int fun(int arg)
{
   ---
}
int (*ptr)(arg type);
ptr = &fun;
```

Eg:

```
int sum(int a, int b)
{  return (a+b);
}
int mul(int x, int y)
{  return (x*y);
}
main( )
{  int (*ptr)(int, int);
   int r1, r2;
   ptr = &sum;
   r1 = ptr(20, 30);  r2 = (*ptr)(30,40);
   printf("\n Value of ptr is %d", r1);
   printf("\n Value of 2nd way  ...
```

```
ptr = &mul;
r2 = mul(...);
r3 = ...(2,3);
printf("\n ... %d", r2);
printf("\n ... %d", r3);
}
```

Call by Reference

In this technique, the address of the actual arguments in the calling function are copied to the formal arguments of called function, using the actual addresses we can have an access to the actual arguments and hence any changes made to manipulate them in the formal to the actual arguments ... function are reflected in the calling function.

Eg: swap 2 variables

```
void swap(int *x, int *y);
main( )
{  int a=10, b=20;
   printf("\n Values before swapping ");
   printf("\n a = %d \t b = %d", a, b);
   swap(&a, &b);
   printf("\n Values after swapping ");
   printf("\n a = %d \t b = %d", a, b);
}
void swap(int *x, int *y)
{  int t;
   t = *x;
   *x = *y;
   *y = t;
}
```

Void Pointer (or) Generic Pointer

A void pointer is a pointer variable declared using the keyword void with the syntax

void *vpname;

Eg: int a=15;
float b=25.34;
void *vptr;
vptr=&a; [illegible]

Value in vptr is a void pointer

When a pointer variable is declared using keyword void it becomes a general purpose pointer and can be assigned address of any variable of any data type, [illegible] a void pointer [illegible] known as generic pointer.

We need to typecast the pointer [illegible] as void pointer has no datatype. This is because a void pointer has no datatype associated with it and there is no way the compiler can know what type of datatype is pointed to by void pointer.

here *(int *)p [illegible]

Eg: #include<stdio.h>

```
main()
{   int a=10;
    float b=55.15;
    void *p;
    p=&a;
    printf("\n value of a using vptr is %d", *(int *)p);
    p=&b;
    printf("\n value of b using vptr is %f", *(float *)p);
```

Pointer to Pointer (or) Double Pointer

(105)

We know that pointer points to a location in memory i.e. pointer is used to store the address of a variable. When we define a pointer to a pointer, the first pointer is used to store the address of the second pointer.

Syntax: datatype **dpname;

Eg: #include<stdio.h>

```
main()
{
    int a=10;
    int *p, **dp;
    p=&a;
    dp=&p;
    printf("\n the value stored at a location is %d", &a, p);
    printf("\n the value of p is %u", p);
    printf("\n the value stored at memory location pointed by p is %d", *p);
    printf("\n the value of double ptr is %u", dp);
    printf("\n the value stored at memory location pointed by dp is %d", **dp);
}
```

dp → p → a (10)

Pointer to Structure

We can have a pointer to structure where a pointer variable can point the address of a structure variable.

Syntax : struct structname *ptrvariable;

The members of the structure are accessed using pointer to structure variable in 2 ways as:

1) using dot operator

Syntax :- (*ptrvariable). member

2) using arrow operator

Syntax : ptrvariable -> member;

Ex:

```
#include<stdio.h>
main( )
{
    struct student
    {
        int rno;
        char name[20];
        float attend;
    }S={1, "Ram", 75.5};
struct student *p;
p=&S;
printf("\n Details of student using dot operator");
printf("\n Roll No: %d", (*p).rno);
printf("\n Name : %s", (*p).name);
printf("\n Attendance : %f", (*p).attend);
```


```
printf("\n Details of student using dot operator");
printf("\n Roll no: %d", (*p).rno);
printf("\n Name : %s", (*p).name);
printf("\n Attendance : %f", (*p).attend);
printf("\n Details using arrow operator:");
printf("\n Roll No : %d", p->rno);
printf("\n Name : %s", p->name);
printf("\n Attendance : %f", p->attend);
}
```

Self referential structures

A structure containing a member which is a pointer to the same structure is called as self referential structure. A self referential structure consists of two fields i.e data and link field.

This is very useful in creating data structures like linked lists, trees etc.

Ex:

```
#include<stdio.h>
main( )
{
    struct student
    {
        int rno;
        char name[20];
        float attend;
        struct student *next;
    };
```

```
struct student s1 = {1, "....", ..};
struct student s2 = {2, "....", ..};
struct student s3 = {3, "....", ..};

s1.next = &s2;
s2.next = &s3;
s3.next = NULL;
printf("\n details of student 1");
printf("\n Roll no: %d", s1.rollno);
printf("\n Name : %s", s1.name);
printf("\n Attendance : %f", s1.attendance);
printf("\n details of student 2");
printf("\n Roll no : %d", s1.next->rollno);
printf("\n Name : %s", s1.next->name);
printf("\n Attendance : %f", s1.next->attendance);
printf("\n details of student 3");
printf("\n Roll no : %d", s1.next->next->rollno);
printf("\n Name : %s", s1.next->next->name);
printf("\n Attendance : %f", s1.next->next->attendance);
```

Dynamic Memory Allocation

We can allocate memory for a variable in two ways as

1) Static allocation
2) Dynamic allocation

Static allocation: In this, memory is allocated for all the variables before run time at compile time.

eg: int a[5]; → at compile time compiler allocates 10 bytes to store 5 elements of the array.

int a=10, *b;

b = &a;

→ In this the address of a is pointer, here compiler allocates 2 bytes at compile time.

Defects: The allocated memory for a variable at compile time can not be altered i.e. explicitly we cannot increase (or) decrease the size of the memory block allocated in static allocation.

Dynamic allocation

If the allocation of the memory is at runtime (or) execution time, we call it as dynamic allocation.

In C lang dynamic memory allocation can be done using 4 library functions as,

1) malloc()
2) calloc()
3) realloc()
4) free()

} defined in stdlib.h

malloc() – this stands for memory allocation. This function allocates exactly as a single block of memory.

Syntax: ptr = (type cast*) malloc(size in bytes);

Eg:- int *p;

p = (int *) malloc(20);

(or)

→ p = (int *) malloc(5 * sizeof(int));

The first line allocates a block of [illegible] to store 5 integer elements. If [illegible] is allocated [illegible] address of first [illegible] pointer p. [illegible] byte

Here the allocated memory [illegible], we access it through pointer variable p only

NOTE:- The dynamic memory allocation is only possible for pointer variables. Since the allocated memory has no name & the allocated memory [illegible] we access the memory through those pointers.

Eg:- find sum of all the elements in a list using malloc()

```
#include<stdlib.h>
main()
{
    int n, *p, i, s=0;
    printf("Input array size:");
    scanf("%d", &n);
    p = (int *) malloc(n*sizeof(int));
    if(p==NULL)
    {   printf("\n Memory allocation failed");
        exit(0);
    }
    printf("\n Enter the elements of the array:");
    for(i=0; i<=(n-1); i++)
    {
        scanf("%d", p+i);
        s = s + *(p+i);
    }
    printf("\n Sum of all the elements in the array is %d", s);
}
```

calloc() :- This stands for contiguous memory allocation. This function allocates multiple blocks of memory having same size which are initialized to zero.

Syntax : ptr = (typecast) calloc(n, sizeof(type));

Eg: int *p, i, n;

p = (int *) calloc(5, 10);

⇒ This allocates 5 blocks of memory each of 10 bytes in size and initializes to zero.

⇒ program to add all the elements in list using calloc()

```
#include<stdio.h>
main()
{
    int *p, n, i, sum=0;
    printf("Input array size:");
    scanf("%d", &n);
    p = (int *) calloc(n, sizeof(int));
    if (p == NULL)
    {
        printf("memory allocation failed");
        exit(0);
    }
    printf("\n Enter array elements:");
    for(i=0; i<n; i++)
    {
    scanf("%d", p+i);
        s = s + *(p+i);
    }
    printf("\n sum of all the elements = %d", s);
```

realloc() :- This function is used to modify or change the previously allocated memory using malloc() or calloc() function.

Syntax : p = realloc(p, newsize);

Eg:
```
#include<stdio.h>
main()
{
    int *p, i, size, nsize;
    printf("Input array size:");
    scanf("%d", &n);
    p = (int *) malloc(size * sizeof(int));
printf("\n the memory location allocated are:");
for(i=0; i<size; i++)
printf("%u\t", p+i);
printf("\n Input new size of array:");
scanf("%d", &nsize);
p = realloc(p, nsize);
printf("\n the newly allocated memory locations are:");
for(i=0; i<nsize; i++)
printf("%u\t", p+i);
    free(p);
}
```

free() :- this is used to release the space allocated using malloc() & calloc() function with the syntax,

free(ptrvariable);

Advantages of Pointers

1) access of data structures as arguments to another function

2) passing a array to other functions value from the functions

3) help to be build complex data structures like trees, graphs etc

4) provides reduce the program execution time by increasing execution speed

Dangling Pointer

A pointer pointing to a memory location that has been deleted (freed) is called dangling pointer.

eg:- int *ptr = (int *) malloc(sizeof(int));
free(ptr); ptr is dangling ptr
ptr = NULL; dangling ptr is cured

Memory leaks

When memory is allocated dynamically, it should be freed when no longer required. If not, memory leak occurs.

When dynamic allocation is no longer required, we should free up the memory otherwise memory from the program will be lost and we end up having a problem known as memory leak.

So when ever we use malloc() & calloc() function to allocate memory, after its use, we should free it using free() else we have a problem known as memory leak.

Files

Advantages

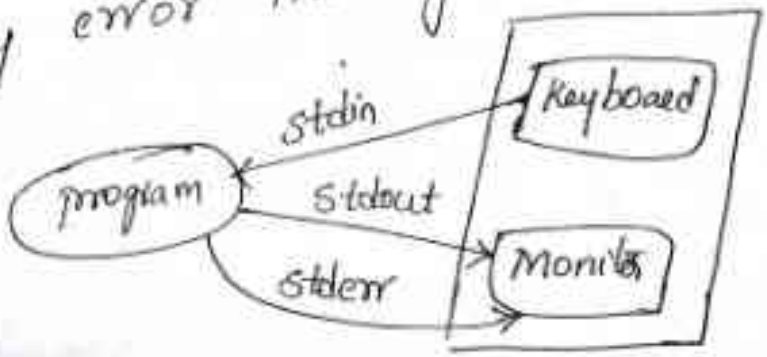
*) files store large amount of data easily
*) files store data permanently.

Streams

→ C language view each file as a sequen
- ce of bytes and each file ends with EOF marker.
→ when a file is opened, a stream is associated with it.
→ A stream is a sequence of characters flowing from one direction to another, and it provides communication channel b/w program and files.

The following streams get connected auto
- matically when a program begins the execution

*) stdin (standard input) stream :- this enables program to read data from keyboard
*) stdout (standard output) stream :- this enables program to display data on monitor/screen
*) stderr (standard error) stream :- enables program to display error messages.



Operations on files

When a program needs to use a file, it needs to perform the following operations

i) Creating and opening a file
ii) Writing data to a file
iii) Reading data from a file
iv) Appending new content to an existing file
v) Closing a file

File opening modes

Mode	Meaning
read (r)	opens a file for reading only
write (w)	a file is created or opened only for writing. If the file already exists, opening the file of overwrite its existing contents by the new contents
append (a)	to append the new content at the end of previous data in a file
r+	reading & writing
w+	writing & reading
a+	adding & reading

Declaring a file pointer

Syntax of declaring a file pointer is,

FILE *filepointername;

Eg:- FILE *fp;

↳ data structure defined in stdio.h

Creating & opening a file

The library function fopen() is used to create and open a file. If the file is successfully created or opened, it returns memory location to file pointer, otherwise it returns NULL pointer.

Syntax: fptr = fopen("filename", "openingmode");

Eg:- FILE *fptr;

fptr = fopen("ascii.txt", "r");

↓ file name ↓ opened in read mode

FILE *map;

map = fopen("b.dat.txt", "w");

↓ filename ↓ opened in write mode

In C language to work with files we have file handling built in (or) library functions defined in <stdio.h> header file as,

function name	Purpose / Meaning
fopen()	Creates / opens a file
fclose()	closes a file
fgetc()	reads a character from file
fputc()	write a character to file
fgets()	reads a string from a file
fputs()	writes a string (character) to a file
getw()	reads an integer value
putw()	writes an integer value
fscanf()	reads formatted data
fprintf()	writes formatted data
fseek()	moves file ptr to ... function
ftell()	tells the current position of cursor
rewind()	from current position ...

(random access functions)

Reading data from a file

1) fgetc() : used to read a character from a file with the syntax,

identifier = fgetc(fp);

Eg: ch = fgetc(fp);

reads a character at a time from the file opened or created by file pointer fp and assigns it to ch (syntax to getchar)

2) fgets() : used to read a string from a file with the syntax,

fgets(varname, no. of characters, fp);

Eg:
```
char str[30];
FILE *fp;
fp = ("file.txt", "r");
fgets(str, 10, fp);
```
(string name) (no. of characters) → read a string from the file pointed by fp - file

3) getw() : (integer) this is used to read a whole number from a file with the syntax,

identifier = getw(fp);

Eg: n = getw(fp);

8) fscanf() - used to read a variable of in different types as a letter from a file with the syntax,

fscanf(fp, "%format specifier", &v1, &v2, ...);

writing data in the file

9) fputc() : used to write a character in file with the syntax.

fputc(variable, fp);

Eg: char ch;
fputc(ch, fp);

10) fputs() : used to write a single string into a file with the syntax,

fputs(variable, fp);

Eg: char str[x] = "hello";
fputs(str, fp);

11) fprintf() : used to write an integer value to a file with the syntax,

fputc(variable, fp);

Eg: int n;
fputc(n, fp);

12) fprintf() : used to write a variable of in different types in a file with the syntax,

fprintf(fp, "control string to be displayed", v1, v2, ...);

closing a file

The syntax is open should be closed using fclose() with the syntax.

fclose(file name);

Eg: fclose(fp);

fcloseall() is used to close all the files at a time which are opened in a program.

Some remaining basic file operations

It is possible that an error may occur during the operation on a file like

1) trying to read beyond the end of file
2) reading from a file which is not open
3) opening a file with invalid filename
4) attempting to write to a write protected file etc.

An unchecked error may result in an abnormal termination of the program or incorrect output.

We have two error handling functions to take care of the above problem while performing I/O operations on files as,

i) feof() :- used to test for an end of file condition in a file

Eg:-
```
if (!feof(fp))
{
 —
}
```

ii) ferror() :- indicates the status of the I/O operation performed on the file

Eg:-
```
if (ferror(fp) != 0)
{
 ===
}
```

Types of files

a) Text files :- contains only the text information — such as alphabets, digits & special symbols.

b) Binary files :- produced as a result of compiling text files — object files, executable files etc.

Copying contents of one file to another

Note :- Before typing the program, create a new file as first.txt and type some data

first.txt

```
// filecopy.c
#include<stdio.h>
main()
{
    char ch;
    FILE *fp1, *fp2;
    fp1 = fopen("first.txt", "r");
    fp2 = fopen("copy.txt", "w");
    if (fp1 == NULL || fp2 == NULL)
    {
        printf("File error in opening");
        exit(1);
    }
    while (!feof(fp1))
    {
        fscanf(fp1, "%c", &ch);
        fprintf(fp2, "%c", ch);
    }
    fclose(fp1);
    fclose(fp2);
}
```

→ Merge two files into another.

Note :- Create two files as m1.txt & my.txt and type some content

M1.txt — my.txt

Vi fmerge.c

```
#include <stdio.h>
main( )
{
    FILE *fp1, *fp2, *fp3;
    char ch;
    fp1 = fopen(" m1.txt", "r");
    fp2 = fopen(" my.txt", "r");
    fp3 = fopen(" merge.txt", "w");
    if(fp1 == NULL || fp2 == NULL || fp3 == NULL)
    {
        printf(" file cannot be opened");
        exit(0);
    }
    while(!feof(fp1))
    {
        fscanf(fp1, "%c", &ch);
        fprintf(fp3, "%c", ch);
    }
    fclose(fp1);
    fclose(fp3);
    fp3 = fopen(" merge.txt", "a");
    while(!feof(fp2))
    {
        fscanf(fp2, "%c", &ch);
        fprintf(fp3, "%c", ch);
    }
    fclose(fp2);
    fclose(fp3);
```

→ Compare contents of two files & print the contents where they differ.

Note : Create two files as comp.txt and comp1.txt and type [illegible]

comp.txt — comp1.txt

vi filecompare.c

```
#include <stdio.h>
main( )
{
    FILE *fp1, *fp2;
    char ch1, ch2;
    int flag=0;
    fp1 = fopen(" comp.txt", "r");
    fp2 = fopen(" comp1.txt", "r");
    if(fp1 == NULL || fp2 == NULL)
    {
        printf("file cannot be opened");
        exit(0);
    }
    while(!feof(fp1) || !feof(fp2))
    {
        fscanf(fp1, "%c", &ch1);
        fscanf(fp2, "%c", &ch2);
        if(ch1 != ch2)
        {
            printf("\n %c is the character pointed by fp1 & %c is the character pointed by fp2", ch1, ch2);
        } flag=1;
    }
    if(flag == 0)
    printf("\n Both files have same content");
```

→ Read a file & to count the no. of characters, words & lines in it.

Note - create a file count.txt and type some content on it.

```
// filecount.c
main()
{
    FILE *fp;
    char ch;
    int nc=0, nw=0, nl=0;
    fp = fopen("count.txt", "r");
    if (fp == NULL)
    {   printf("\n file cannot be opened");
        exit();
    }
    while (!feof(fp))
    {   ch = fgetc(fp);   - reading a character from the file
        if (ch == EOF)
        break;
        nc++;
        if (ch == ' ' || ch == '\t' ...
            || ch == '\n')
        nw++;
        if (ch == '\n')
        nl++;
    }
    printf("\n No. of characters count = %d", nc);
    printf("\n No. of words count = %d", nw);
    printf("\n No. of lines count = %d", nl);
```

Storage Classes

A storage class defines the scope and life time of variables (i.e. it tells us how we can access a variable). The following things are known with storage class as

1) the variable scope (lifetime)
2) the location where the variable will be placed.
3) the initial value of the variable
4) lifetime of a variable

+ A variable along with datatype, is also associated with a storage class.
→ In C language we have four types of storage classes as,

1) automatic
2) register
3) static
4) extern

Automatic storage class - keyword auto is used to declare an auto storage class. The scope of an auto variable is limited to the block in which it is declared (i.e. the content from one set of the defined { }). Once the control exits the block, the variable in that block to which it is access is lost in the defined.

register storage class
register

Static storage class

static is the key-word to declare static storage class.

They are stored in main memory and the uninitialized value of a static variable is [illegible] initialized to zero.

Eg:- Syntax :- static datatype [illegible];

static int i;

```
#include <stdio.h>
void display( );
main( )
{  display( ); — 1
   display( ); — 2
   display( ); — [illegible]
}
void display( )
{ static int i=1;
     printf("%d\n", i);
     i=i+1;
}
```

Note: the static [illegible] [illegible] [illegible] the value [illegible] [illegible].

(2)

extern storage class

keyword extern is used to declare an external variable.

The extern variables are shared among calling and called functions i.e they should belong to the [illegible] function.

i.e the variable are accessed throughout the program, they are also called as global variables. the default value of an extern [illegible] is zero.

Syntax :- extern datatype variable;

Eg: extern int a,b;

```
#include <stdio.h>
void display( );
int n=5;
main( )
{
    printf("value of n in calling fn = %d", n); —5
    n++; — [illegible]
    display( );
    printf("\n value of n after calling fn = %d", n);
}
void display( )
{
    n++; — [illegible]
    printf("\n value of n in called fn %d", n);
}
```

```
main( )
{
   extern int n=5;
   [illegible] — [illegible]
}
```

Pre-processor directives

In C language, a preprocessor is a text substitution tool & it instructs the compiler to do required pre-processing before actual compilation.

Before compiling, source code is processed by a program called preprocessor by compiler. The process is called pre-processing. Commands used in pre-processor are called preprocessor directives which begin with # symbol.

C language has following list of preprocessor directives as:

preprocessor	Meaning
#include	mostly used in header file inclusion. Syntax: #include<filename.h>
Macro	define macro constant value of any type. Syntax: #define macro_name value; #undef ...
Conditional compilation	compiler with respect to condition. Syntax: #ifdef ..., #if, #else ...
Other directive	#undef, #pragma etc.

User defined header files

Like how we have system defined header files using file inclusion directive, #include we can also define our own header files with the syntax:

#include "filename.h"

```
#include <stdio.h>  -> system defined
#include "myheader.h"  -> user defined
```

myheader.h

```
int fact(int n)
{
    int i,f=1;
    for(i=1;i<=n;i++)
    f=f*i;
    return f;
}
```

file1.c

```
#include <stdio.h>  -- sys
#include "myheader.h"  -- own
main()
{
    int f;
    f=fact(5);  --> function fact gets linked from myheader.h
    printf("\n factorial of given no is %d", f);
}
```

Single linked list representation:

Each node contains only one link which points to the next node of the list.

Double linked list

Each node contains 2 link fields [illegible] list

Circular linked list

Each field of the last node contains [illegible] to the first node.

Command line Arguments

So far we have given main() as a function without arguments. The [illegible] main() can have two arguments. [illegible] [illegible] argc which stands the argument count and argv which stands the argument vector [illegible] the system.

int main(int argc, char *argv[])

- count of no. of arguments passed to main()
- [*] char *argv[]: array of strings to hold the values.

[illegible] main() is not the [illegible] function that [illegible] passing values to other functions [illegible] main() [illegible] to pass values to [illegible] to pass values to main() [illegible] the prototype [illegible] because [illegible] command line [illegible] the values are called as command line arguments.

Note :- 1) argc - denotes the number of arguments to be passed

2) argv - [illegible] a pointer array

```
#include<stdio.h>
int main(int argc, char *argv[])
{   int i=1;
    if (argc != 3)
    {
        printf("Arguments passed thru command
                line not equal to 3");
        return 1;
    }   for (i=1; i<3; i++)
printf("\n Program name : %s", argv[0]);
printf("\n argv[%d] : %s", [illegible]
```

Command line arg program (w/ [illegible])

```
#include <stdio.h>
int main(int argc, char *argv[])
{
    if (argc != 4)
    {
        printf("Arguments passed thru command
                line are not equal to 4");
        return 1;
    }
    printf("\n Program name : %s", argv[0]);
    printf("\n first arg value : %s", argv[1]);
    printf("\n Second arg value : %s", argv[2]);
    printf("\n third arg value : %s", argv[3]);
}
```

[illegible] pass arguments in [illegible] 3 arguments

[illegible] ⇒ [illegible] [illegible] argv[0] argv[1] argv[2] [illegible]

After compiling and executing the program run [illegible] pass arguments thru command prompt as shown above.

*) Always in argv[0] we have program name followed by arguments given to the program [illegible] in Linux/unix [illegible]

Macro

```
#include <stdio.h>
#define max a>b?a:b;
 (max name, a>b?a:b value)
main()
{
    int a,b,m;
    printf("Input values for a & b:");
    scanf("%d%d", &a, &b);
    m = max;
    printf("%d is max among
            %d and %d", m, a, b);
}
```

$\Phi = \cot x - \operatorname{cosec} x$

$\Phi = \cot^{-1} x - \operatorname{cosec}^2 x$

$\frac{d\Phi}{dx} = \dots$

#undef

a = 5

b = 15

5 > 4

```
#include <stdio.h>
int max(int, int); — prot.
main()
{ int a,b,m;
  printf("Input values for a & b:");
  scanf("%d%d", &a, &b);
  m = max(a,b); — fn call
  printf("\n %d is max among %d and
          %d", m, a, b);
}
int max(int x, int y) — def
{
    return(x>y ? x : y);
}
```