BTech CSE
1st Year Sem1

Full notes PDF

Utkarsh Aggarwal

PROGRAMING FOR PROBLEM SOLVING

(1) (Using) (Programming for problem solving) using (C) 29/Nov/2022,
(Turbo C++) By = Shalu Ma'am,

Unit-1 (Basics of Computer)

Computer:

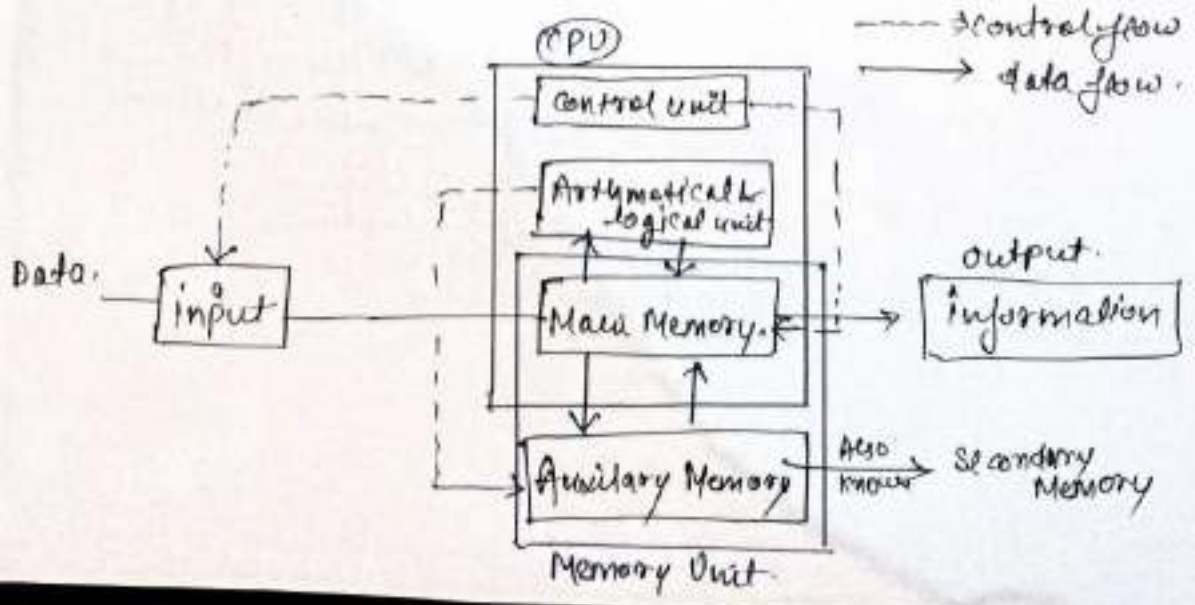
- "Charles babbage" is called the father of computer.
- The computer is an electronic device that takes input from the user and processes the data under the control of the set of instructions called program. And gives the result (output).
- it can process both arithmetic & logical operations.

Input → Process → O/P. (Result.)

(L) for Sure. (10 Marks)

Block diagram of a computer & its components:

- The diagram that illustrates the primary components of the computer system is known as Block diagram of computer.
- A Block diagram of a computer displays its structural representation of computer system.
- The block diagram gives you a quick overview of the working process of a computer from inputting the data to retrieving of data (desired one). The following diagram represents a block diagram of computer system.



- A computer System is a combination of 3 components :-
  - 1) input & output unit
  - 2) CPU
  - 3) Memory.

Input Unit

- This unit consists of input devices such as: mouse, keyboard etc.
- These devices are used to input information or instructions into the computer system.
- The input unit performs the following major functions:-

1) This unit converts the input data / instructions into binary form for further processing.
2) This unit transmits the data to the main memory of the computer.

Central Processing Unit (CPU)

- CPU or Central Process Unit is known as the brain of computer system.
- its an electronic hardware device, that processes all the operations (Arithmetic/logical) of the computer
- The CU/ALU are jointly called the CPU.

Control Unit (CU)

- This controls all the activities & operations of the computer.
- it is also responsible for controlling input/(op), memory and other devices connected to the CPU.
- The CU acts like a supervisor which determines the sequence in which programs, instructions are executed.
- This determines the sequence of operations to execute the given instructions.

Arithmetic & logical Unit (ALU)

- The data given(input) by input device is stored in the Primary storage unit.
- The (ALU) performs arithmetic & logical operations.
- The Arithmetic unit controls simple operations such as add^n, subtraction, division & multiplication.
- On the other hand the (LU) controls the Logical operations AND, OR, EQUAL, >, <.
- Apart from it the (LU) also responsible for performing several other operations such as comparing, selecting, matching & merging data.

The CPU performs following major functions :-

- The CPU controls all the components, software & data processing of the computer System.
- The CPU takes data from I/P devices, executes the data & sends out the output devices.
- The CPU processes all the operations including all the arithmetical & logical operations.

Memory Unit (MU)

- Its an essential part of the computer system which is used to store data and instructions before & after processing.
- There are 2 types of (MU) :-

① Primary / Internal / Main :- This memory is also known as the volatile memory. The data stored in it is temporary.

→ data will get lost, if PC is turned off.

② Secondary / Auxiliary / External :- It called as non volatile

- Data stored in it, is permanently saved.
- The data stored in it is safe even when their is a power failure or no power supply.

  e.g - harddisk is usually 2nd memory.

Note:- Primary memory is the only memory that is directly accessible to the CPU, 2nd memory is not directly accessible to CPU.

- Data accessed from the secondary memory is first loaded into RAM and then further transfered to the processing unit.

Output Unit

- This unit consists of devices that are used to display the results.
- o/p data is first stored in memory & then displayed in human readable form through o/p devices.
- Some of the widely used o/p devices are Monitor, Printer, projector.

Summary

① The data is Accepted by the I/P unit its given by the user its then converted into binary form for computer to read it.

② Binary form for the comp to read. The info is then send to the MU for storage & processing.

③ The reqd data that needs to be processed is accessed by the CPU. It's accessed from primary memory. The AU & LU operations are then performed on the data. The control unit schedules all the activities for the smooth working of the computer.

④ The data is sent to the storage unit, it's used for storing or further processing purposes. ⑤ The o/p unit receives the final proccessed o/p.

PPS NOTES | 01-Dec-2022 |

[Generations of Computers]

* (1st Generation)
- Eckert & Mauchly developed the 1st electronic computer.
  ENIAC in 1946 in USA.
  (Electronic numeric integrated & calculator).
-

it consists: Vacuum tubes, Punch cards, Paper tapes, Batch OS

- During the 1st Gen computers whose developed using vaccum tubes as the core technology.
- The time period of 1st gen computer was 1946-1959.
- Input was based on punch cards & paper tape and o/p was displayed on printout.
- limited internal storage capacity, slow I/p & o/p operation
- Consumes lots of electricity, its not portable.
- The operating System used in 1st gen is Batch OS.
- 1st Gen computers used the machine language the lowest level programming language. so that it could be easily processed & understood by computer.

* (2nd Generation)
- This gen computers are IBM700, ATLAS, KL1901.
- the 2nd gen began with widespread use of transistors.
- The time period of 2nd gen was from 1954-1965.
- Magnetic tapes & disk was used for 2nd storage. magnetic core technology was used for primary memory.
- During 2nd gen comp used assembly languages instead of Binary machine language.
- high level languages FORTRAN → formula translation, COBOL → common business oriented langauge
- Reqd less Power supply to operate.
- There where batch processing & multiprogramming OS used.

01/Dec/2022 PPS Notes.

(3rd Generation)

- This Gen computer are IBM360, NCR395, TDC-316 developed by Jack Kilby.
- 3rd Gen of comp was characterised by the use of (IC) in computer rather than transistors. (Integrate of circuit)
- Time period = 1965–1971, the Integrated circuits are small in size & helped to make computer, smaller than its predecessor.
- Magnetic disk was used for secondary storage.
- Besides advanced I/p & o/p device such as mouse, keyboards and monitor were introduced in this generation.
- In terms of language 3rd Gen computers used higher level languages such as FORTRAN (II - IV) PASCAL, ALGOL-68, COBOL.
- In this generation development of mini computers,
- In this generation real time & Multiprogramming OS are used.

02/Dec/2022

(4th generation)

- This generation computers are DEC10, STAR1000, etc.
- During this generation computer were developed using the microprocessors as the core technology.
- Microprocessors were also based on (LSI) (large scale integration & (VLSI) (Very LSI).
- They were developed by assembly of several IC's on a single silicon chip.
- The time period (1971 – 1985)
- Microprocessors not only helped making computers smaller but also made them so powerfull & reliable. Due to this compact size computers became available for personal use during this time.
- OS used are network OS, Distributed OS.

• HLL such as C, C++, Java, Python, JS etc were used in computer of this generation. 2/Dec/2022 (IPS)

(5th Generation)

• This gen computers are desktop, laptop etc
• Time period of this gen (1985 → till date)
• this gen PC are based on (ULSI) (ultra LSI), AI softwares & parallel processing hardwares
• In this gen of PC are particularly based on (HLL) such as C, C++, Java, dotnet, Python, JS, Ruby
• RAM & ROM is used for primary memory.
• HDD, SSD is used for secondary memory.
• OS windows all versions., LINUX, UNIX, UBUNTO

[Summary]

Generations	Core element	Time Period
1st Generation	Vacuum tubes	1946 – 1959
2nd Generation	Transistors	1959 – 1965
3rd Generation	IC's	1965 – 1971
4th Generation	LSI & VLSI	1971 – 1985
5th Generation	ULSI	1985 - till date.

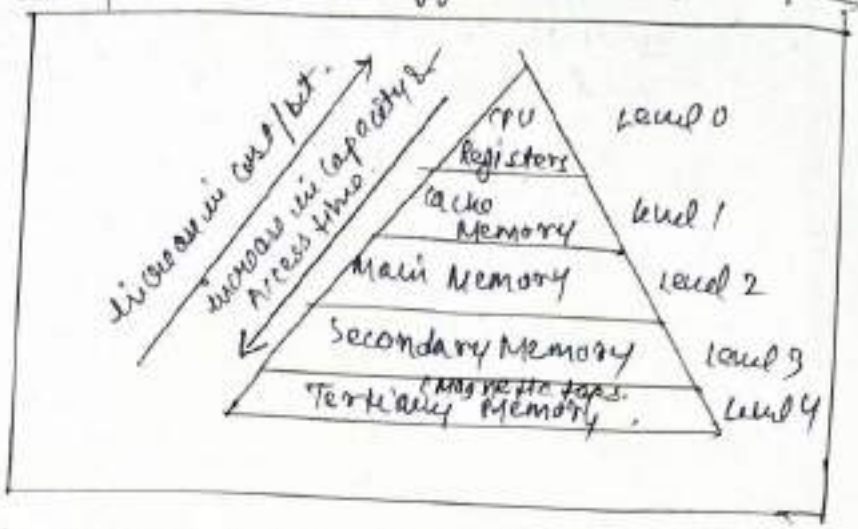
5/Dec/2022 # [Memory & its heirarchy] # (V. Imp for CT)

*(Memory):- • PC memory is just like a human brain
• its used to store data/information & instructions
• PC memory is the storage space in the computer where data is to be processed & instructions reqd for it are stored.

• The computer memory heirarchy looks like a pyramid structures.

• Memory heirarchy is an enhancement to organise the memory such that it can minimise the excess time.

The figure below clearly demonstrates diff levels:- (05/Dec/2022)



(Level 0)

CPU registers:- • This memory is the smallest & fastest memory in the computer. It is not a main memory part and is located in CPU in the form of registers which are the smallest data holding elements.

• It holds instructions that are currently processed by the CPU.
• All data is reqd to pass through registers before it can be processed.

Cache memory: (Level 1):

• This is a very high speed memory it acts as a buffer b/w the CPU & main memory.
• Used to hold those data & programmes which are most freq used by CPU.

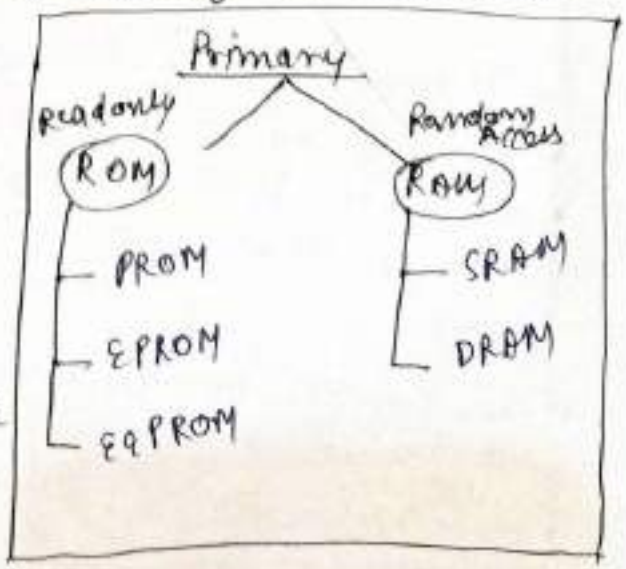
Primary Memory (Level-2):-

• Also known as main memory of PC. Its used to store data & programming or instructions during PC operation.

• Its of 2 types:-

① RAM (Random Access Memory)

• Its a volatile memory.
• It stores info based on power supply.
• If the power supply fails all the data & info on this will be lost.
• It temporary stores programs/data which has to be executed by the processor



(a) SRAM :- • it uses transistors & circuits of this memory are capable of retaining their states as long as the power is applied.

• This memory consists of the no of flip flops with each flip flop storing one (1 bit).

• it has less access time. hence its faster.

• It doesn't need to refresh memory contents.

(b) DRAM :- • it used capacitors & transistors and stores the data as a charge on the capacitors.

• it needs to refreshed.

(2) ROM (read only memory).

• it's a non-volatile memory. it stores info even when the failure of power supply.

• ROM is used to store information that's used to operate the system.

• We can only read the programmes & data that is stores on it.

Types of ROM:-

1) PROM:- (Programable ROM) :- • it's a PC memory chip & its possible to program it once after creation.

• After Programming the PROM the info we write to it becomes permanent.

2) EPROM:- (Erasable Programable ROM) :- we can erase & re-program eprom without replacing it.

• its an extension of PROM where you can erase the content of rom. by using UV rays.

3) EEPROM: (Electronically EPROM) :- • the written contents can be erased by electricity.

06/Dec/2022 PPS

Secondary storage:- (level 3)

Its also known as auxiliary memory & backup memory, its a non-volatile memory and used to store a large amount of data/ info. Its stored in secondary memory is permanent and it is slower than primary memory.

* Types of fixed storage *
- Hard disk drive (HDD)
- Solid state drive (SSD)

* Types of removable storage *
- DVDs
- Memory Cards
- Magnetic tapes
- Pen drives.

Tertiary storage:- (level 4)

tertiary storage device like Magnetic tapes are present at level 4 they are used to store removable files and are the cheapest & largest in size.

[Input/Output Devices]

* input unit
- we need to first enter the data & instruction in the PC, before any computation begins.
- this task is accomplished by the input devices.
- the data accepted is in a human readable form.
- The input device converts it into a computer readable form

* Input Device

the input device is the means through which data & instruction enter in a PC.

① Camera:- its used in live meeting/conference.
- it transmits picture from one PC to other. or can be used for recording purposes

② keyboard:- it's a device used to input letters or numbers into different places/programs. it have special keys to help user to operate PC.

PPS 06/Dec/2022

③ Mouse:- its used to open & close files, navigate websites and click on a lot of commands when using different apps.

④ Microphone:- its used to record sound. its then saved as sound file on PC.

⑤ Scanner:- its used to copy picture or other things & save them as files.

⑥ Joystick. its used to move the cursor from place to place and to click on various items. mostly used in games.

*Output Devices

① Monitor:- its the screen on which words, numbers, graphics can be seen. the monitor is the most common output device.

② Printer:- it prints whatever is on the monitor onto paper like words, numbers, pictures.

③ Speakers:- its gives you sound output from your PC some speakers are built into the PC.

④ Headphones. they give o/p from the PC. they are similar to speakers except they are worn on ears. only one person can hear at a time.

PPS seminar 6/Dec/2022

(Operating System & its functions)

* **Operating System:-** It's a collection of software that manages computer hardware resources and provides common services for computer programs.

- An OS acts as an intermediary between the user of PC and PC hardware.
- Purpose of OS is to provide an environment in which a user can execute programs conveniently & efficiently.

* **functions of OS:-** • **Process management**

- every software that runs on computer whether in the background or in frontend is a process.
- process management is an execution unit in which a prog/task operates
- when more than 1 process runs on the system, the OS decides, how & when a process will use CPU. hence the name called CPU scheduling

* **CPU scheduling Algorithms are:-**

① FCFS (first come first serve) ③ SJF (Shortest job first)
② RR (Round Robin) ④ Priority based scheduling.

• **Memory Managment** • it's the management of the main or the primary memory whatever program is executed it has to be present in the main memory.

- Main memory is a quick storage area that may be accessed directly by the CPU.
- The OS has to keep track of how much memory has been used and by whom.
- it has to decide which process needs the memory space & how much.

* File management :-

- It manages all the file related activities such as organization, storage, naming, sharing, retrieval & protection of any file.
- The files on a system are stored in different directories.
- Keeps records of the status & locations of files.
- Allocates & deallocates resources.
- Besides who get the resources

07/Dec/2022

* Device Management :-

- An OS regulates device connection using drivers,
- the process may require devices for their use this management is done by OS.

* Storage Management :-

- Systems have several levels of storage which includes primary storage, secondary storage & cache storage.

* Security :-

- This module protects the data & information of a computer system against 'VIRUS' & unauthorised access.

* I/P management :-

- Any kind of input that you want to give to the PC goes through the OS.
- Any o/p that we get is also acted upon by the OS.

* Command interpretation :-

- You as a user input your commands in a language that is understood by you.
- Since the PC only understand Binary Language that is 0 & 1 the OS manages the interpretation of the CMD & then guides the resources to process them.

(Language)

07/Dec/2022 PPS

its a mode of communication that is used to share ideas, opinions with each other.

For eg:- if we want to teach someone we need a language that's understood by both communicators.

* (What's the Programming language) *

- Its a computer language that" used by programmers to communicate with PC.
- its a set of instructions written in any specific language C, C++, Java, Python, JS, Ruby to perform a specific task.

(Types of Programming languages)

* (low level Program language) (LLL)

- It is a machine dependent programming language.
- the processor runs low level programms directly without the need of compiler or interpreter. so the programs written in (LLL) can be run by us very fast.
- (LLL) is further divided into 2 parts:

① Machine language:- its a type of (LL) its also called as machine code 'or object code'

② Assembly language:- its also a type of (LL) that is designed for specific processor.

- it represents the set of instructions in a symbolic & human understandable form.
- it uses short pnoumonics code for instruction.
- it uses an assembler to convert the assembly to (ML) machine language

07/Dec/PPS

* (High level language) * (HLL)

- (HLL) is designed for developing user friendly software progs and websites.
- this language requires a compiler/interpreter to translate prog into (HL).
- the main advantage of (HLL) is that its easy to read, write, maintain.
- (HLL) includes C, C++, Java, Python, JS, Ruby, dotnet, CSAB,

* (Middle level language) * (MLL)

- Middle level langugs is a type of programming language that has features of both (LLL) & (HLL).
- it is known as the intermediate level programing langugs.
- FORTRAN:- formula Translation, use engineering & scientific in the year 1957.
- ALGOL:- Algorithm language 1958 use for calculation.
- COBOL:- common Bussiness oriented language 1959, uses specially bussiness apps.
- BASIC:- Beginner All Purpose symbolic instruction code language 1964 use students computer solving Problem
- CPL:- Combined Programming language 1963.
- BCPL:- Basic combines Programming language 1967.
- PASCAL:- Programming language named for Blaise Pascal, 1910 use:- Business & scientific purposes.
- C:- this programming language 1972 developed by Dennis Richie.

* (Translator) * → converts source code to Machine code.

- Compiler
- Interpreter
- Assembler

PPS (08/Dec/2022)

(Compiler)

It is the system software that stores the completes program, scans it, then translates the whole program into object code (MLL) and then creates an executable code.

- it translates entire code in a single sem.
- Compilation is a process of converted HLL into MLL with the help of structure that called compiler and whole process is called compilation.
- Key combination for compiling C-program is ALT+F9

object file:- The machine level code generated by compiler is known as object file.

- Key for executing C-programming is [ctrl+f9].

Interpreter.

It is a system software used to translate high level language source code into MLL (machine) object code line by line.

- Takes each line of code and converts it into machine code & stores it into the object file.
- Interpreted program takes a long time in executing.
- To overcome this disadvantage especially for large program compilers were developed.

(9th/Dec/2022)

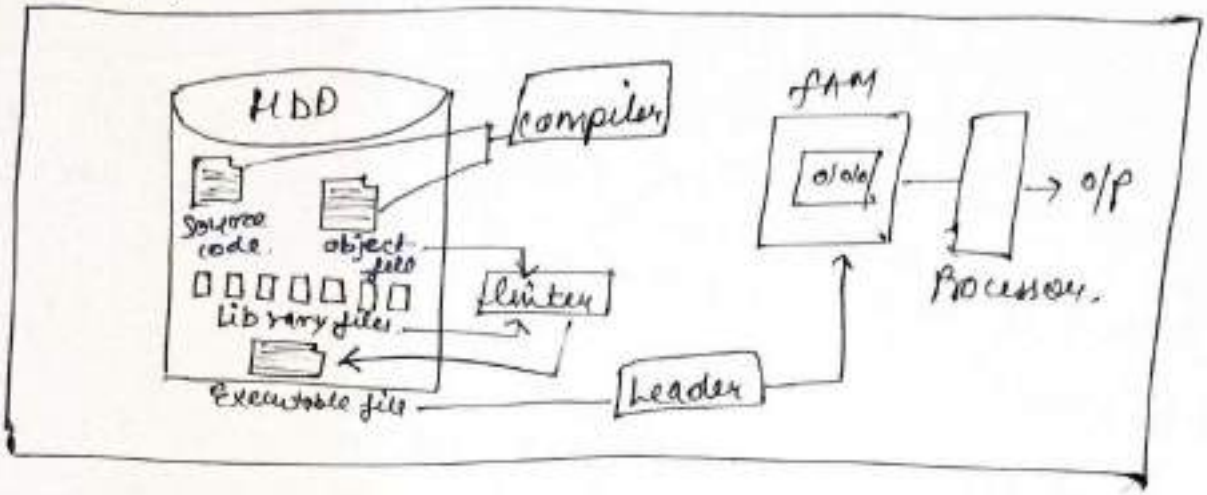
(linker & loader)

* (linking of C) *

its the process of attaching together with other program files & function that are needed by the program with the help of software that is linker, & this processing is called linking.

linker is a software which takes input as object file & library files & produces executable files

<u>Executable files</u> :- combination of object file & library file



<u>Loader</u> :- It is a software which takes executable file from secondary memory and place it into the primary memory. hence when the program is in the primary it would send all the instruction present in the program to processor & produces the required o/p

(<u>Structure of C Program</u>)

Documentation section.
link section
Defination Section
Global declaration section
main() { declaration part Executable part } → local variable
Sub program section
Sub program function [name of the function] function1 function2 ⋮ function n.

12/Dec/2022 | 12/Dec/2022 | (PPS)
PPS

```
extra;
void main ()
{ fun(), disk();
printf( )
}
void fun()
{
}
void disk()
{
}.
```

[function call.] — called user functions into main().

[* If you'll not assign value to a it will print the garbage value.]

Define RT 3.14.

(user defined function)

[* global variable is written outside {} and main().]

(Documentation Section)

- This section consists of the description of the program, name of the program and the creation data & time of programm.
- It's specified at the start of the Program in the form of comments.
- Documentation can be represented as for:-

  Single line:- //- - - - - - -

  Multiple line:- /*- - - - - - - - - - - - - - */

(Link Section)

- This section contains all the header files used in a programme it informs the system to link the header files to the system library. It's given by

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

(Definition section)

- All the symbolic constants are written in this section.
- Symbolic constants are called as Macros.
  Macros is a segment of code which is replaced by the value of Macro.
- Macro is defined by [# define directive]

* (Object like Macros) *

- This macro is an identifier that is replaced by value.
- its widely used to represent numeric constant.

for ex:-

```
# define PI 3.14
# define MAX 100
```

7. Here PI is the macro name which will be replaced by the value :- [3.14]

(Global declaration)

- its a variable that is declared outside the function or block. that means you can use these variables throughout the program. all global variable are retained in this section.
- The Global variable can be used anywhere in the program. this section also defined the user defined function.

for eg:-

```
int a=10;
main ()
{ printf(" ")
}
void function()
{ printf(" ")
}
```

(Main ())

- its the major section from where the execution of the program begins.

12/Dec/2022 (PPS)

- The main section is values declaration & executable section.
- Main() :- The declaration section is the part where all the variable are declared.
- Both the declaration & execution part are inside the { }.

Syntax

```
void main()
{ declaration part
  executable part
}.
```

(Example code below)

```
void main()
{ int sum;
  sum = 3 + 5;
  printf("Result is = %d", sum);
}.
```

(Local variable)

- The variable that is declared inside the function or block is called local variable.

Eg :-

```
void main()
{ int a;
  clrscr();
  printf("%d", a);
  if(a != 0)
  { int b = 10;
    printf("%d", b);
  }
  getch();
}.
```

(Sub Program Section)

* This section contains user defined functions which are called by main(), when reqd.

Ex = fun1
fun2
fun n

13/Dec/2022

* Semantic error
= Meaningless error

(Errors in C)

* Errors are the problems or the faults that occur in code, which makes the behaviour of the program abnormal and experienced developers can also make these faults.
* Programming errors are known as bugs & the process of its removal is known as debugging.
* These errors are detected by the time of compilation or execution.
* These errors must be removed from the program for the successful execution of the program

(Types of errors in C)

① Syntax errors:- • Also known as compilation errors as they occured at the time of compilation.
• These errors are easily debugged.

commonly done errors:-

- if we miss parenthesis [ ] { }
- Displaying value of a variable without declaration
- missing semicolon at the end of statement.

PPS 13/Dec/2022

Syntax error

```
#
#
int main()
{ a = 10;
printf(" ")
return 0;
getch();
}.
```

Runtime error

```
#
#
int main()
{ int a = 2;
int b = 2/0;
Printf("The value of b is %d", b);
getch();
```

Linker error

```
#
#
int main()
{ int a = 78;
Printf("the value of
a is %d", a);
return 0;
getch(); }
```

Semantic error

```
#
#
int main()
{ int a, b;
a = 2;
b = 1;
a+b = c;
return 0;
getch();
```

Logical error (VVIMP)

```
#
#
int main()
{ int i, sum = 0; // variable initialisation
int k = 1;
for(i = 1, i<=10; i++); // logical error as we put the
                          semicolon after loop.
{ sum = sum + k;
k++; }
Printf("The value of sum is %d", sum);
getch()
}.
```

② Runtime errors:-

- Some errors exist during the execution time even after successful compilation are run time errors.
- When the program is running & it's not able to perform the operation is the main cause of the time errors.
- These errors are very difficult to find as the compiler does not point to these errors.

③ linking errors

- it's mainly generated only when the executable file of the program is not created.
- it can happen due to wrong function prototyping or usage of wrong header file.

④ logical errors:

- The logical error is an error that leads to an individually undesired product/output.
- These errors interfere & creates inccorect o/p but they are error free.

⑤ Semantic errors:-

These are the errors that occured when the statements are not understandable by the compiler.

(Data types)

- In C Programming data types are declarations for variables. this determines the type & size of data associated with variables.
- A data type specifies the type of data that a variable can store such as integer, floating, character etc.

PPS

DATA TYPES

- Primary or Basic
  - int
  - char
  - float
  - double
- Secondary or Derived
  - Array
  - Pointer
  - Structure
  - union
- Enumeration
  - ↳ Enum
- Void
  - ↳ void.

- The basic data types are integer based and floating point Based.
- C language support both signed & unsigned literals or modifiers.
- The memory size of basic data types may change A/C to 32/64 bit OS. varies from compiler to compiler.
- Below list follows 32 Bit OS. Compiler.

Data type	keyword	Memory bytes	Range.	format specifier
• integer.	Short int	2 Bytes	-32,768 to 32,767	%. Rd
	unsigned Short int	2 Bytes	0 to 65535	%. hv
	int/long int	4 byte	-2147483648 to +2147483647	%. d.
	unsigned int/ long int	4 byte	0 to 4294967295	%. v
	long long int	8 byte	-(2^63) to (2^63)-1	%. lld
	unsigned long long int	8 byte	0 to 18446744073 709551615	%. llu
Character.	signed char.	1 byte	-128 to 127	%. c
	unsigned char	1 byte	0. to 255	%. c
float	float	4 byte	1.2E -38 to 3.4E+38	%. f.
Double	Double	8 byte	1.7E -308 to 1.7E+308	%. lf
long double	long Double	16 byte	3.4E-4932 to 1.1E+4932	%. lf

① int:- these are whole no that can have both 0, +, -ve value but no decimal value.

for eg = 0, -5, 10.

② char:- keyword char is used for declaring character type variable.

for eg =

char test = 'k'; (char value are surrounded by single quotes eg 'a' or 'b')

③ float & double:- float and double are used to hold real no

Eg:- float salary;
double price;

* In C, floating point numbers can also be represented in exponential.

for eg:- float factor = 22.44e2;

④ void:- * It means nothing or no type.

* if a function is not returning anything its return type should be void.

⑤ short and long:- if you need to use a large no, you can use a type specifier long.

Eg = long a;
long long b;

⑥ signed & unsigned:-

- in C, signed & unsigned are type of modifiers
- You can alter the data storage of a data type by using them:-

* signed :→ allows for storage of both positive and -ve no.

* unsigned → allowed for storage of only +ve no.

Variables in C

- A variable is a name of memory location.
- It stores data.
- It's value can be changed & it can be reuse of many times.
- It is a way to represent memory location through symbol so it can be done easily.

(# Declaring (creating) variables #)

Syntax: Data type variable name = value;
int a = 12

*(Rules for defining variables)

- A variable name must only contain alphabets, digits & underscore.
- It must start by alphabet or underscore.
- no start by digits, no white space allowed within.
- cannot be any reserved keyword.

* (Classification on basis of scope)

(1) local variable (2) Global variable.

* (classification on the basis of storage classes) *

- Static, Automic
- External, Register.

* Constants:- It is a value or variable that can't be changed in the program.

*(2 ways to define constant in C) *

- Const keyword
- # define pre processor.

Types of Constant

Constant	Example
① integer constant	10, 20, 450 etc.
② character constant	'a', 'b', 'x' etc.
③ Real or floating Point constant	10.3, 20.2, etc.
④ string constant	"Ankit"

using constant keyword

```
#
#
void main()
{ const int a = 100;
  clrscr();
  Printf("%d", a);
  getch();
}
```

O/P = 100

for character:-

```
#
#
void main()
{ const char a = 'A';
  clrscr()
  printf("%c", a);
  getch();
}
```

O/P = A.

for floating Point

```
#
#
void main()
{ const float a = 3.14;
  clrscr();
  Printf("%f", a);
  getch();
}
```

O/P = 3.14000

for string:

```
#
# void main()
{ const char a[10] = "Ankit";
  clrscr();
  Printf("%s", a);
  getch();
}
```

O/P = Ankit

define

①
```
# include <stdio.h>
# define PI 3.14
  main()
{ printf("%f", PI);
}
```

O/P = 3.140000

②
```
# include <stdio.h>
# define
  MIN(a,b) ((a)<(b))?(a):(b))
void main()
{
Printf("Minimum b/w 10 & 20
  is : %d\n"
  MIN(10,20));  }
```

O/P = minimum b/w 10 & 20 : 10

(Keywords in C)#

- A keyword is a reserved word,
- can't be used as variable name, const name, etc.
- There are 32 in total C

1) Auto
2) do
3) go to
4) signed
5) void
6) break
7) double
8) if
9) size of
10) while
11) case
12) else
13) int
14) static
15) unsigned
16) char
17) enum
18) long
19) struct
20) volatile
21) const
22) extern
23) register
24) switch
25) continue
26) float
27) return
28) typedef
29) default
30) for
31) short
32) union.

(operator :)

- An operator is simple a symbol that is used to perform operation.
- There can be many types of operations like Arthmetic, logical or bitwise etc.
- operator operates the operands.

for eg = (a) + (b) operator of Add^n

operands

(Types of Operator)

- Unary → uses only one operand
  - increment
  - Decrement
  - size of operator
- Binary
  - Arthmetic
  - Relational
  - logical
  - Bitwise
  - Assignment
- Ternary
  - Terary or Conditional operator.

operators	Type
++, --	unary operators.
+, -, *, /, %	Arethmetic.
<, <=, >, >=, ==, !=	Relational operator.
&& , ||, !	logical operator.
&, |, <<, >>, ~, ^	Bitwise operator.
=, +=, -=, *=, %=	Assignment
? :	Ternary/Cond^nal.

(Associativity table)

Operator.	Description.	Associativity
( )	Parenthesis	
[ ]	Brackets	
.	Member selⁿ via obj name	
→	" via Pointer.	
++ --	Postfix inc/Dec Prefix inc/Dec	left to Right Right to left.
+ - ! ~	Plus, minus (unary) logical negation/ bitwise complement	Right to left
*	Pointer	Right to left
&	Reference (Address)	
* / %	Multiplication/Division /modulus	left to Right
+ -	Addition subtraction	
<< >>	Bitwise shift left, right	
<=	Relational <=	
>=	Relational >=	
== !=	Relational = !=	
&	Bitwise And	
^	Bitwise exclusive OR	
|	Bitwise inclusive OR.	
&&	logical And	left to Right
||	logical OR	
?:	Ternary Conditional	Right to left
=	Assignment	
,	comma.	→ left to Right

PPS

* (Use of operator precedence & Associativity etc) *

→ Operator Precedence :

* it determines which operation is performed first in an expression with more than one operators with different precedence.

for eg :- 10 + 20 * 30

first yeh hogi due to high precedence.

first yeh

→ Operator Associativity :

its used when 2 operators have same precedence in an expression. Associativity can be R to L or L to R.

Eg :- 100/10 * 10

left right

* operators precedence & Associativity are 2 char of operators that determine the evaluation order of sub expressions in absence of ( ).

for eg :- 100 + (200/100) - (3 * 10)

(I) (II)

100 + 2

(III)

30

102 - 30

(IV)

72 Ans

Operators	Associativity	PPS Precedence
( ) . → (++ --)	left to right	1 (high priority)
(++--) + - ! ~ * &	Right → left	2
* / %] Arthmetic op.	left → right	3
+ +] Arthmetic op.	left to Right	4
<< >>	L → R	5
<<= >>=	L → R	6
= = ! =	L → R	7
^ (Bitwise XOR)	L → R	9
& (Bitwise AND)	L → R	8
| (Bitwise OR)	L → R	10
&& (logical and)	L → R	11
|| (logical OR)	L → R	12
? :	R → L	13
=, +=, -=, *=, /=, %=, etc.	R → L	14
,	L → R	15 (least Priority)

(PPS) (19/Dec/2022)

* Arithmetic Operator in C

Its used to perform mathematical operation such as addition, subtraction, multiplication & division, modulous on the given operater.

$5+3=8$
$5-3=2$
$2\times4=8$ etc

* Addition operator

It is a simple (+) operator used to add to given operands. we can use (+) operator with different data types such as integer, float, long, double, enumerated & string type data to add the given operator. eg = $C=A+B$

* **Minus operator:-** its denoted by (-) ve sign. Data type of the given number can be of different types such as integer, long double etc in the programming language.

* **Multiplication operator:-** its represented by (*) & used to return the product of 2 numbers.
- Data type of given no can be of different types. such as int, float & double C language. eg = c = A *B;

* **Division operator:-** it's an arithmetic no operator that divides the first no by 2nd no. using different division operator we can ÷ the integer, float, double & long data type variables. eg = c = A/B

* **Modulus operator:-** its represented by (%) sign & is used to return the reminder by (÷) the first no by the 2nd num. c = A% B

(Relational operators use)

① Equal to operator (==)

Syntax: operand 1 == operand 2

Eg: int a=5; int b=10; { o/p a==b=0 5 is not equal to 10 }

② Not Equal to operator (!=)

Syntax: operator 1 != operator 2

Eg: int a=5; b=10; { a!=b ① 5 is not equal to 10 }

③ less than operator (<)

Syntax: A<B

Eg: A=45 B=89 { 45<89 }

④ Greater than operator

Syntax: A>B

eg: A=45 B=32 { o/p 45>32 }

⑤ less than Equal to operator (<=)

Syntax: A<=B

Eg: A=45 B=45 { value of A is equal to B 45=45 }.

⑥ greater than equal to operator (>=)

Syntax: A>=B

Eg: A=28 B=79 { value of B is greater than A (79>28) }

(logical operator in C) PPS (19/Dec/2022)

operators.	Description
① && (logical AND)	• This operator returns true if all relation statement combined with && are true, else it returns false.
② || (logical OR)	• The operator returns true if at least one of the relational statement combined with (||) is true else it returns false.
③ ! (logical Not)	• This operator returns the inverse of the statement's result.

Ex:1 (&& logical AND)

If any gets false it will be whole false

A	B	A && B
false	True	false
false	false	false
True	True	True
True	false	false

case1 = 10>20 && 20>5
false.

case2 = 10<20 && 20<5
false.

case3 = 10>20 && 20<5
false.

case4 : 10<20 && 20>5
True.

Note :- The AND operator only given the true value.

20/Dec/2022

Ex.2 (OR logical ||)

A	B	A||B
True	True	True
false	True	True.
True	false	True.
false	false	false.

case1 = 10<20 || 20<5
we'll get true.

case2: 10>20 || 20>5.
True.

case 3: 10<20 || 20>5
True.

case4: 20<10 || 20<5
false.

if any one gets true
Answer gets true.

Ex-3 Logical Not operator (!)

condition	! condition
1 (True)	0 (false)
0 (false)	1 (True)
Case1	Case2
!(12>20) we will get true	!(10<20) we will get false

(Bitwise operator in C)

operators	Meaning of operator
&	Bitwise AND operator
|	"OR"
^	"Exclusive OR"
~	one's complement
<<	left shift
>>	Right shift

(Truth table)

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

PPS (20 Dec 2022)

(One's complement)

x	$\sim x$
0	1
1	0

(Bitwise & Operator)

7 → 0 1 1 1
4 → 0 1 0 0

4 → 0 1 0 0

```
#include <stdio.h>
# include <conio.h>
int main()
  { clrscr();
    int a = 7, b = 4;
printf(" The o/p of Bitwise AND operator a &b is %d", a&b);
    return 0;
    getch();
  }.
```

* Bitwise OR operator:

7 → 0 1 1 1
4 → 0 1 0 0

7 → 0 1 1 1

7/4 = 7.

* Bitwise NOT operator

7 → ~0 1 1
8 → 1 0 0 0

* Bitwise 1's operator

1 → ~0 1 1 1
8 → 1 0 0 0

(Ternary Operator)

Syntax:- x>y ? x : y

```
#include<stdio.h>
#include<conio.h>
int main()
{ clrscr();
  int x, y, max;
  Printf("Enter the values of x & y");
  scanf("%d %d", &x &y);
  max = x>y ? x:y;
  Printf("larger of %d and %d is %d", x, y, max);
  getch();
    }.
```

[maam didn't used clrscr();]

```
#include<stdio.h>
#include<conio.h>
  int main()
{ clrscr();
  int x=3, y=4, z;
  z = (x==3 ? (y==4 ? 6:8) : 0);
  Printf("%d\n", z);
    return 0;
    getch(); }.
```

O/P = 6

(Typecasting)

PPS (21/Dec/2022)

- its a way to convert a variable from one data type to another data type.

[int stores = 2-4 byte
Double stores = 8 byte]

Syntax:-

(Data type) expression
or
(type name)

* explicit is used for precise Ans *

* Typecasting is of 2 types:-

→ i) [implicit / Automatic conversion] Its done by the compiler and there is no loss of information.

Eg = let x is 8 bit
y is 16 bit
now perform addition x+y.

(ii) [Explicit typeconversion :] - Its done by the programmer and there will be a loss of info.

Eg = float x = 3.435;
int y = (int)x;

[Program to show typecast execution]

```
# include <stdio.h>
# Include <conio.h>
  main()
{ int a=5, b=2;
  float c;                    — (2) 0
  c = a/b;
  Printf("value of c%f,"c);
c = (float)a/b ;              — (2.5)
  Print("After typecasting value of c%f,c);
  getch()
  }.
```

① int a=10, b=3;
float c=a/b;

② int a=10, b=3
float c = float (a/b); 3.0

③ int a=10, b=3;
float c = (float) a/b; 3.33

CT/ST/sem final

(conditional Branching) # (VVIMP) (23/Dec/2022)

(i) If statement

Syntax:
```
if(condition)
{ // code to be executed
}
```

Example:
```
# include <stdio.h>
# include <conio.h>
int main()
{int marks;
clrscr();
Printf (" Enter Marks");
Scanf ("%d," &marks);
if (marks >= 250)
{printf ("Pass");
}getch();
}.
```

(ii) If else Statement:-

Syntax:-
```
If (expression)
{ // statement
}
else
{ // statement
}
```

Example:

```
#include <stdio.h>
# include <conio.h>
{ int marks;
  clrscr();
printf ("Enter marks");
scanf ("%d", &marks);
  if (marks >= 250)
  { printf ("Pass");
  }. else
     printf ("fail");
     getch();
     }.
```

(iii) Nested if else statement. → printf not used

syntax:-

```
if (expression)
{ if (expression)
   { statement block 1;
   }
   else
   { statement block 2;
     }
     else
   { statement block 3;
     }
```

Example:-

```
#include <stdio.h>
# include <conio.h>
int main ()
{ int marks = 245, attendance = 23;
  clrscr();          printf(" ")
                     scanf(
  if (marks >= 250)
    { printf ("Pass");
      }
  else printf ("sorry your attendance is low");
```

```
}
else
Printf("fail");
    getch();
    }.
```

(V) if else if ladder (Most VIMP)

Syntax:-

```
if(condition 1)
{
  // statement
  }else if (condition2)
  { // statements }
  else if (condition 3)
  { // statements }
  else { // statements
        }
```

Example:-

```
# include <stdio.h>
# include <conio.h>
int main ()
{ int marks;
    clrscr();
      Printf("Enter marks");
      Scanf("%d", &marks);
        if (marks >=300)
        { printf("First");
          }
        else if (marks >=250)
          { printf("Second");
        else if (marks >=143)
          { printf("Third");
          } else
        Printf("fail");
          getch(); }.
```

PPS (23/Dec/2022)

(Use of break & Default with switch)

Syntax

```
                    int, char
switch (expression)
{
 case 1: statement 1;
         break;
 case 2: statement 2;
         break;
   |
 case n : statement n;
         break;

 default: Statement;
         Break;
}
```

Example:-

```
#include <stdio.h>
#include <conio.h>
int main()
{ char ch;
printf("Enter a character");
scanf("%c", &ch);
switch (ch)
{
    case 'a';
    case 'e';
    case 'i';
    case 'o';
    case 'u';
    printf("%c is a vowel", ch);
    break;
  default :
    printf("%c is not a vowel", ch);
    break;
    } getch();
}
```

Date: / /
Page No.

→ Continue statement :-

Ex)

```
#
#
{
int i, a, sum = 0;
for(i=1; i<=5; i++):
{
printf("Enter a Integer");
Scanf("%d", &a);
if(a<0)
{
Continue;
}
Sum = Sum + a;
}
printf("Sum = %d", Sum);
}
```

O/P

Enter a integer 1	Enter an integer -1
" " " -5	" " " 2
" " " -6	" " " 3
" " " -7	" " " 4
" " " -8	" " " 5
Sum = 1	Sum = 14

Date: / /
Page No.

Ex
(2)

```
#
#
int main()
{
for(i=1; i<=10; i++);
{
if(i==7)
continue;
printf("%d \n",);
}
getch();
}
```

op (Skip 7)
1
2
3
4
5
6
8
9
10

→ Goto Statement

• Syntax :

Syntax 1	Syntax-2
Forward Jump	Backward Jump
goto label;	label:
—	—
—	—
—	—
label:	goto label;
—	
—	
—	

Date: / /
Page No.

Forward jump

```
#
#
void main()
{
printf ("Hello");
goto label;
printf ("welcome");
label: printf ("C programming");
getch();
}
```

O/P

Hello
C Programming

Backward Jump

```
#
#
int main();
{
int n;
n=1;
repeat:
printf("n = %d", n);
n++;
if (n<=10)
goto repeat;
}
```

O/P

1
2
3
10

C Switch Statement:

- The switch statement in C is an alternate to if-else if ladder statement which allows us to execute multiple operations for the different possibles values of a single variable called Switch variable.
- Here, we can define various statements in the multiple cases for the different values of a single variable.

Date: / /
Page No.

The syntax of switch statement in C language is given below:

```
Switch(Expression)
{
//code to be executed;
break; //optional
Case 2;
//code to be executed;
  break;
  :
  default:
  //code to be executed if all cases are not matched;
}
```

* Rules for switch statement in C language:-

① The switch expression must be of an integer or character type.

② You are not allowed to add duplicate cases.

```
int main()
{
int x=1;
Switch(x)
{
Case 1: printf("x is 1");
      break;
Case 1: printf("x is 1");
      break;
Case 2: printf("x is 1");
      break;
}.
```

→ Error
duplicate case value

Date: / /
Page No.

(3) Only these expressions are allowed in switch which result in an integral constant value.

```
int main ()
{
int a=1, b=2, c=3;
switch (a+b*c)
{
Case 1: printf("choice 1");
        break;
Case 2: printf("Choice 2");
default: printf("default ");
        break;
}
}
```

```
int main() // It returns float values are not allowed.
{
float a=1.15, b=2.0, c=3.0;
switch (a+b*c)
{
```

(4) Float values is not allowed as a constant value in case but only integer constants / constant Expressions are allowed in case label.

```
int main()
{
float x=3.14;
Switch (x)
{
Case 3.14: printf("x is 3.14");   // Produce Error because
        break;                     code 3.14 & case 1.1 is
Case 1.1: printf("x is 1.14");     not allowed
        break;
Case 2: printf("x is 2 ");
        break;
}
}
```

Date: / /
Page No.

(8) int main()
{
int x = 23; // integral constant is allowed.
Switch(x)
{
Case 3+3; printf("choice is 1");
break;
Case 3+4+5 : printf("choice 2");
break;
default : printf("default ");
break;
}
}

(4) Variables expressions are not allowed in case labels although macros are allowed.

(9) int main()
{
int x=2, y=2, z=23;
Switch (x) // Variables Expressions are not allowed.
{
Case y: printf("Number is 2");
break;
Case z: printf(" Number is 23");
break;
default : printf("default case");
break;
}
}

Date: / /
Page No.

(9).
```
# include <stdio.h>
# define Y2
# define Z 23
int main() {
    int x=2;                    //Macros are use
    Switch(x)                    as a case label)
{
Case Y: printf("Number is 2");
        break;
Case Z : printf("Number is 23");
        break;
default: printf("default case");
        break;
  }
}
```

(5) Default can be ~~used~~ placed anywhere inside switch it will still get evaluated if no match is found.

(6):
```
int main()
{
int x=2;                        // default always
Switch(x)                        get Evaluated after
{                                checking all the cases.
default : printf("default case");
        break;
Case 1 : printf("Number is 1");
        break;
Case 2 : printf("Number is 2");
        break;
  }
}
```

Switch case Examples.

Date: / /
Page No.

① Using break Keyword

```
#include<stdio.h>
#include<conio.h>
int main()
{
int i = 2;
Switch(i).
{
case 1: printf("case 1");
    break;
Case2: printf("case 2");
    break;
Case3; printf("case 3");
    break;
Case4: printf("case 4");
    break;
dfault : print f("default");
}
return 0;
getch();
}
```

O/P - case 2.

② Using without break Keyword

```
#include<stdio.h>
#include<conio.h>
int main()
{
int i=2;
Switch(i)
{
Case1: printf("case 1");
Case2: printf("case 2");
Case3: printf("Case 3");
Case4: printf("Case 4");
default : printf("default");
}
return 0;
getch();
}
```

O/P.

Case2, case3 Case 4 Default.

Q:- You can use character in switch case

```
Switch (2+3)
{
Case5: printf("2+3 makes 5");
    break;
Case4: printf("2+3 is not 4");
    break;
}
```

O/P

2+3 makes 5.

Date: / /
Page No.

Eg:

```
# include <Stdio.h>
# include < Conio.h >
int main()
{
Char ch='b';
Switch(ch)
{
Case 'd' : printf("case D");
    break;
Case 'b' : printf(" Case B");
    break;
Case 'c' : printf("Case C");
    break;
Case 'z' : printf("case z");
  break;
default : printf("default");
    }
return 0;
getch ();
}
```

O/P Case B.

* Use of Continue with Example :-

The continue statement works similar to break statement. The only difference is that break statement terminates the loop whereas continue statement passes control to the conditional test. i.e. where the condition is checked, skipping the rest of the statements of the loop.

Date: / /
Page No.

Eg: Using break.

```
for (i=1; i<=10; i++)
  {
  if (i==7)
  break;
  printf("%d\n", i);
  }
getch()
}
```

O/P
1
2
3
4
5
6.

Eg: We can not print those numbers which is divisible by 5.

```
for (i=1; i<=20; i++).
{
if (i%5==0)
continue;
printf("%d\n", i);
}
getch();
}.
```

O/P
1
2
3
4
~~5~~
6
7
8
9
11
12
13
14
16
17
18
19.

Not print:
5, 10, 15, 20
not show.

Goto Statement :

* The goto statement is known as jump statement in C.
* The goto statement can be used to jump from anywhere to anywhere ~~with~~within a function.
* As the name suggests goto is used to translate the program control to a predefined label.

Date: / /
Page No.

* The goto statement can be used to repeat some part of the code for a particular condition.

* It can also be used to break the multiple loops which can't be done by using a single break statement.

Syntax	Syntax 1 (Forward Jump)	Syntax 2 (Backward Jump)
	goto label;	label:
	!	!
	label:	goto label;

* In the above syntax, the first line tells the compiler to goto or jump to the statement marked as label.

* Here label is a user defined identifier which indicate the target statement.

Ex Forward Jump.

```
int main()
{
int i, int sum=0;
for(i=1; i<=10; i++)
{
sum = sum + i;
if (i==5)
goto label;
}
label:
printf("Sum is %d", sum);
}
```

Forward Jump

Label

Backward Jump

Q1 → WAP to check a No is odd or even using goto statements

Eg2

```
# include <Stdio.h>
# include <conio.h>
int main()
{
int a;
printf("\n Enter number");
Scanf("%d", &a);
if(a%2==0)
goto Even;
Else
goto odd;
Even
printf("\n Number is even");
odd
printf("\n Number is odd");
return 0;
getch();
}
```

Forward Jump

Eg3 ~~Backward Jump~~

```
# include <Stdio.h>
# include <conio.h>
int main()
{
int n=1;
Jump:
printf("%d", n);
n++;
if(n<=10)
goto jump;
return 0;
getch();
}
```

Backward Jump

Q2 WAP to "N" Natural numbers:

```
# include <Stdio.h>
# include <conio.h>
int main()
{
int n, i=1;
printf("\nEnter number");
Scanf("%d", &n);
printnum:
printf("\n %d", i);
i++
if(i<=n)
goto print num;
getch();
return 0;
}
```

Label

Backward Jump

Unit-1 Topics (use in Unit-3)

Date: / /
Page No.

1⇒ Algorithm & it's characteristics, Flowchart & it its symbols, Pseudocode :-

⇒ Algorithm :- An algorithm is an effective step by step procedure for solving a problem in a finite Number of steps.

2⇒ Algorithms are very important to the computers prom information. because a computer program is basically an algorithm that tells computers that what specific task to perform in what specific order to acomplish a specific task.)

③ The same problem can be solved with different methods

④ So, for solving the same problems different Algorithms can be ~~defined~~ designed. In these algorithms No of steps, time & effort may very more or less.

* Characteristics of an Algorithm :-

→ An algorithm should have following characteristics:

① Finiteness
② Input
③ Output
④ Definitness
⑤ Effectiveness.

Date: / /
Page No.

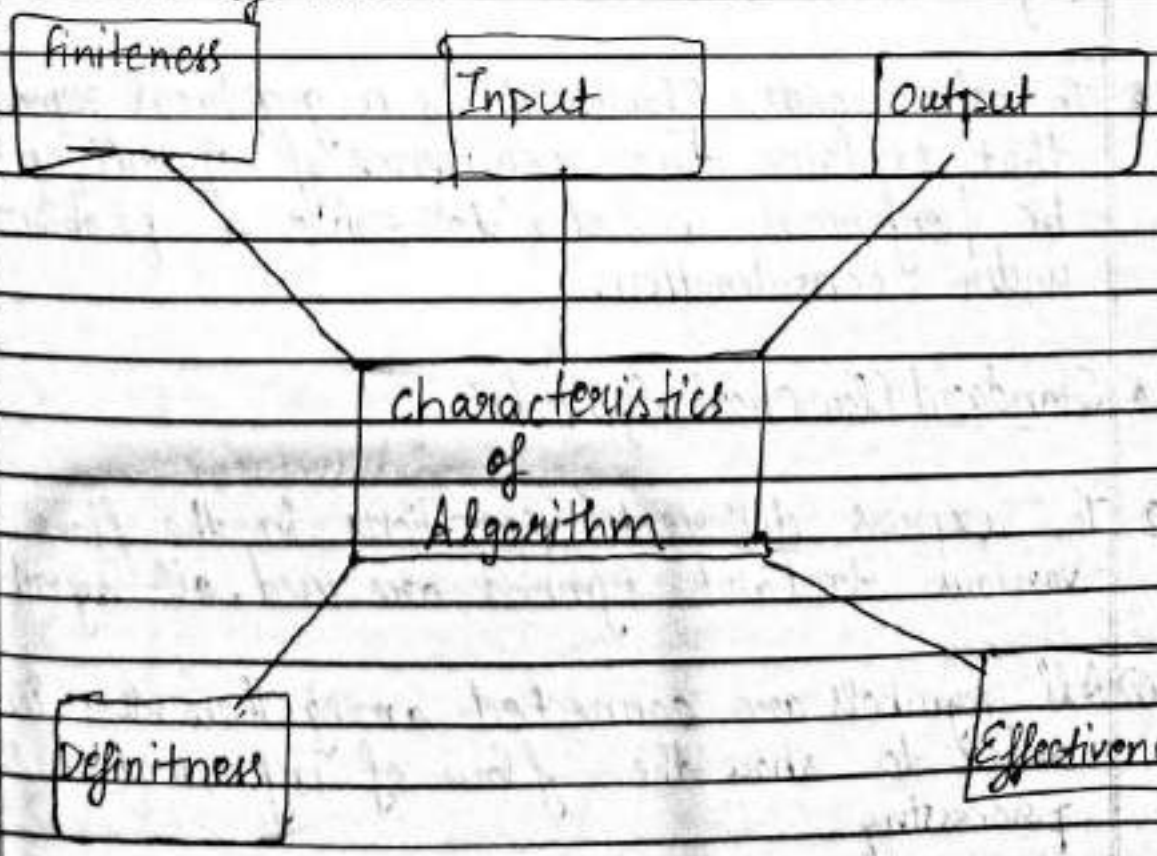
① Finiteness :
An algorithm should have finite No of steps & It should end after a finite time.

② Input :-
An algorithm may have many inputs.

③ Outputs :-
It should result atleast one output.

④ Definitness :-
Each step must be clear, well defined & precise. Their should be No any ambiguity (error).

⑤ Effectiveness :-
Each step must be simple & should take a finite amount of time.



Date: / /
Page No.

* Algorithm of sum of two numbers:

Step 1 = Start

~~Step 2~~ Step-2 Take two variables a, b and store the result in sum.

~~Step 3~~ Step-3 Take input from user of a & b.

~~Step 4~~ Store the result of a & b in sum

~~Step 4~~ Step-4 Check O/P is correct or not

~~Step 5~~ Step-5 End

⇒ Flowchart & it's symbol:

Definition:

① Flowchart is basically a pictorial or diagramatic representation of an algorithm using standard symbol.

② In other words, Flowchart is a graphical repren that explains the sequence of operations to be performed in order to solve a problem under consideration.

* Standard Flow Chart Symbols:

① To express different operations in the flow chart various standard symbols are used. ~~all symbols~~

② All symbols are connected among themselves in order to show the flow of information & processing

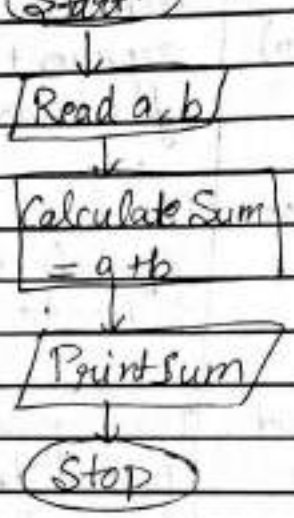
Date: / /
Page No:

Different symbols are prescribed by american National standard institute (ANSI) which are frequently required while drawing flow chart are tabulated below.

Flowchart Symbol	Symbol Name	Description
	Termina (Start or Stop)	Terminals (oval shapes) are used to represent start and stop of the flowchart.
	Flow lines or Arrow.	Flow lines are used to connect symbols used in flow chart & indicate direction of flow.
	Input/output.	Parellograms are used to read input data & output or display information.
	Process	Rectangles are generally used to represent process. Eg:- Arithmatic operations etc.
	Decision or Diamond Symbol	Diamond shapes are generally used to check any condition or take decision for which there are two answers they are ~~Yes~~ Yes (True) or NO (False).
	Connector.	It is used to connect or join flow lines.

Flowchart Symbol	Symbol Name	Description
[	Annotation	It is used to provide additional information about another flow chart symbols in the form of comments or remarks.

* Flowchart of addition of two Numbers:-



* Pseudo code:-

① Pseudo code is an informal way of programming description that does not required any strict programming language syntax.

② Pseudo code is a kind of structured English for describing algorithm.

③ It allows the designer to focus on the logic of the algorithm without being distracted by details of language syntax.

④ It is used in planning an algorithm with sketching out the structure of the program before the actual coding takes place.

Date: / /
Page No.

* Pseudo code of Addition of two numbers:

START
Read a,b
Compute sum = a+b
Print sum
END.

* Write an Algorithm and Pseudo code of celsius to Fahrenheit conversion:

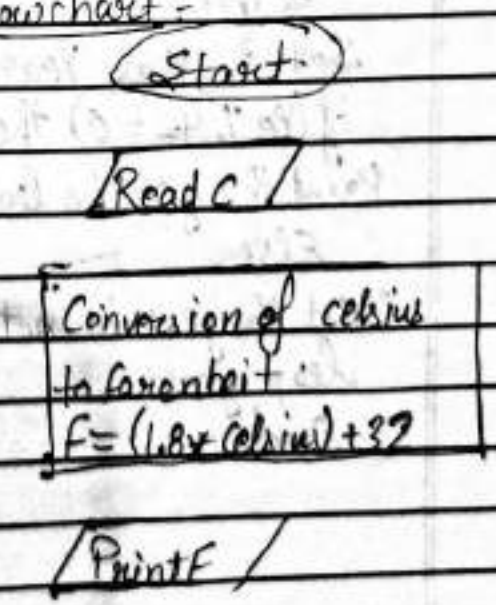
Algorithm:

Step1: START
Step2: Read the value of celsius.
Step3: Conversion of celsius to Fahrenheit
Step4: Print the value F.
Step5: STOP.

* Pseudo code.

START
Read C
Compute F = (1.8 * Celsius) + 32
Print F
END

Flowchart:



Date: / /
Page No.

Q→ Write an Algorithm & pseudo code of Leap year or Not, also draw it's flow chart.

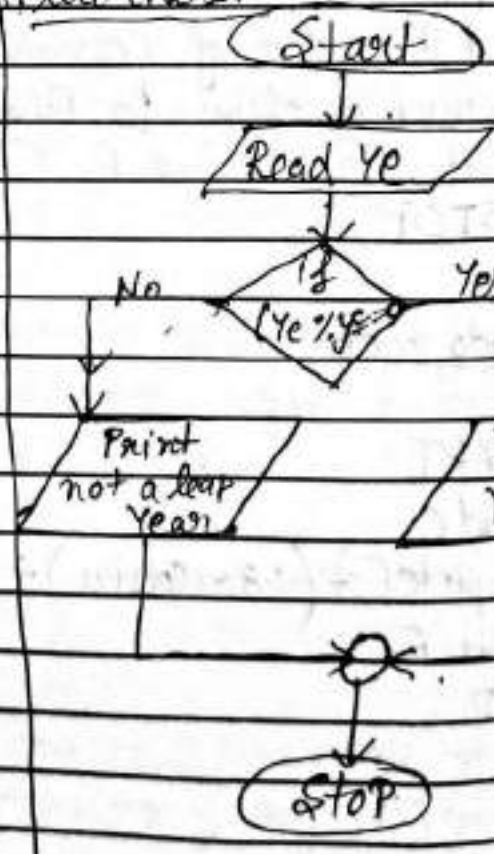
* Algorithm :

Step1: START
Step2: Declare year as int Data type
Step3: Read the Year
Step4: Check if (Ye % 4 = = 0)
Step4.1: Print it is a leap year
Step5: Else
Step5.1: Print it is not a leapyear
Step6: STOP

* Pseudo code.

START
Set initial Year
if (Ye % 4 = = 0) then
Print the year is leap year
Else
Print the year is not a leap year
End.

* Flow chart



Date: / /
Page No.

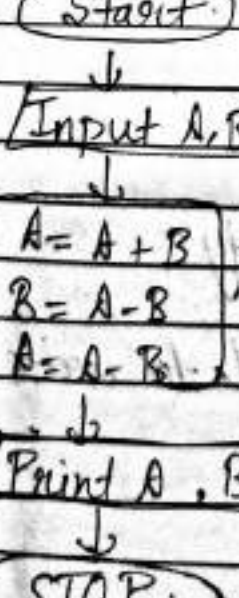
Q:- Write an Algorithm & pseudo code of swapping of two numbers using third varible

* Algorithm

~~START~~
~~Declare No as int Data type~~
~~Read the No.~~
~~Print~~

Step-1 START
Step-2 Declare No. a, b, c as int Data Type.
Step-3 Read two Numbers a and b.
c = a ;
a = b ;
b = c ;
Step-4 Print a and b
Step-5 STOP

Flow chart



Date: / /
Page No.

Module-3
Iteration & Loops

Looping :- The Looping can be defined as repeating the same process multiple times until a specific conditions satisfies it is Known as Iteration also.

Why looping :- The looping simplifies the complex problems into the ~~easy~~ easy ones.

⇒ It enables to alter the flow of the program so that instead of writing the same code again & again, we can execute the same code for a finite No. of same code.
For eg:- If you need to print "university of Calcutta" 10 times than instead of using the printf statement 10 times we can use printf ones inside a loop which runs upto 10 iterations.

* Advantages of looping :-

① It provides code reusability.

② Using loops we do not need to write the same code again & again.

③

Date: / /
Page No.

* Types of C Loops → There are three types of loop in C language.

① While loop.

② do ~~with~~ while loop.

③ for loop.

* while loop :-

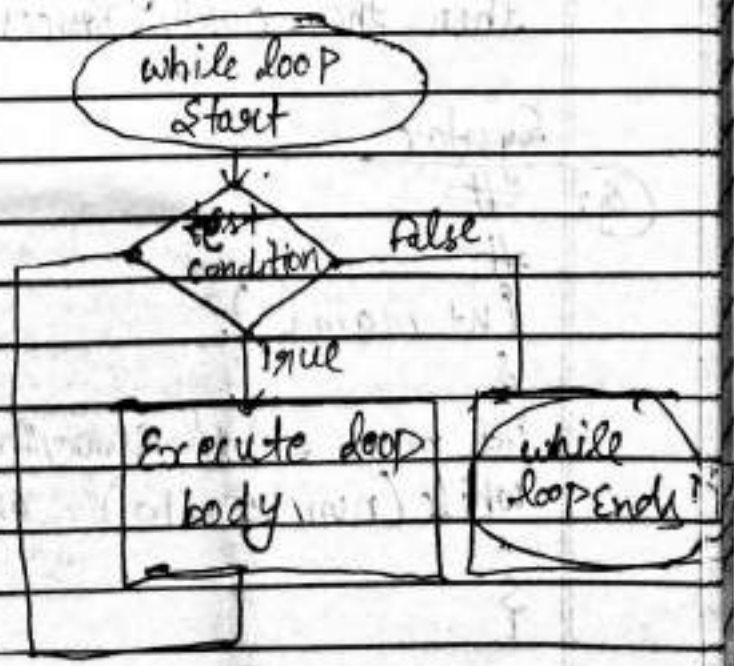
① While loop execute the code until condition is false.
② In while loop a condition is checked before executing the body of a loop.

③ It is also called as a prechecking loop or entry control loop.
④ If a condition is true then and only then the body of a loop is executed.

Syntax :-

```
while (condition)
{
//Statements;
update Expression;
}
```

* Flow chart



Date: / /
Page No.

* Program

```
#include<stdio.h>
#include<conio.h>
int main()
{
int i=2; //initialization Expression
while (i<10) // Test Expression
  2<10)

  { ------> // Body of loop
  Printf ("Helloworld \n");
  i++; //update Expression.
  }
  getch();
  return 0;
  }
```

① In while loop if the condition is not true then the body of a loop will not be even not even once. If it contains only one statement then the curly Braces are not compulsory.

Syntax

```
#
#
int main().
{
int num = 1; // initializing the variable
while (num <=10) //while loop
                 with Condn
{
printf(" %d \n", num);
num++; //increment operator
num++; //
}
getch();
return0;
}
```

o/p
1
2
3
4
5
6
7
8
9
10

Date: / /
Page No.

* do while loop:-

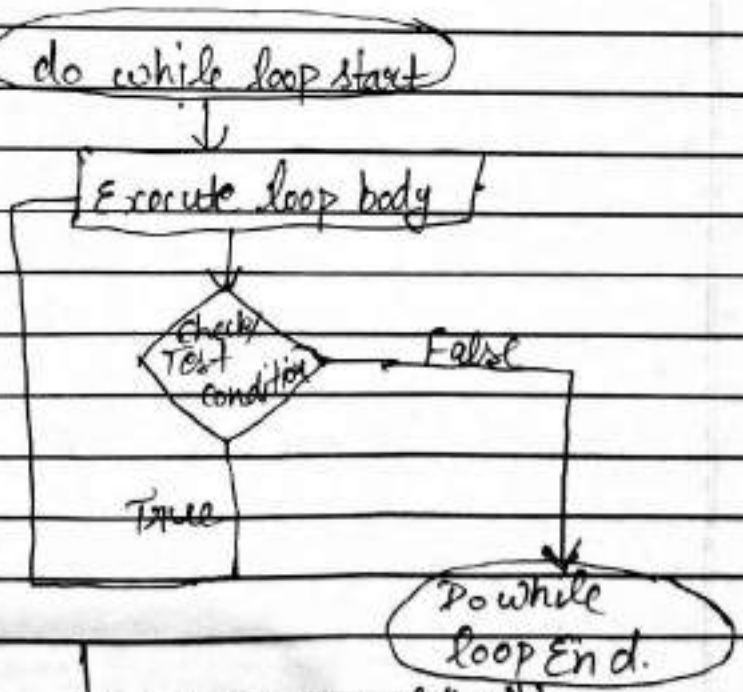
① A do while loop in C is similar do the while loop except that the condition is always executed after the body of a loop. ② in do while loop the test condition is tested or evaluted at the end of the loop body.

③ The loop body is executed atleast ones whether the test condition is true or false. it is also known as exit control loop.

Syntax:

```
do
{
//Statements;
update Expression;
}
while (condition);
```

Flowchart



Program

```
#
#
int main()
{
int i=2;
do
{
printf("Hello world\n");
i++; //update Expression
}
while (i<1); // Test condition
getch();
return 0;
}.
```

// Input 123 -----> 1+2+3 = 6 output.

Sum of digit

```
# include <stdio.h>
# include <conio.h>
int main()
int n, r, sum = 0;
printf("Enter any number : ");
scanf("%d", &n);
while (n > 0) // n = 123
{
 r = n % 10;
sum = sum + r;
n = n/10;
}
printf(" sum of digits : %d", sum);
getch();
return 0;
}
```

$$\begin{array}{r|l} 10 & 123 \\ & 120 \\ \hline & 3 \end{array} \quad 12 \qquad r = 3$$

$$\begin{array}{r|l} 10 & 123 \\ & 120 \\ \hline & 3 \end{array} \quad 12 \qquad n = 12$$

Reverse a number

// Input 234 - - - → 432 will be output.

```
# include <stdio.h>
# include <conio.h>
int main ()
{
 int n, r ;
 printf (" Enter number : ");
 Scanf ("%d", &n);
 while (n>0)  // n = 234
 {
  r = n % 10;
  printf ("%d", r);
  n = n/10 ;
 }
 getch ();
 return 0;
}
```

$r = n \% 10$ → $10\overline{)234}\,(23$, 230, 4 → $r = 4$

$n = n/10$ → $10\overline{)234}\,(23$, 230, 4 → $n = 23$

O/P

Enter number : 234

432

Terminate loop by input negative number

```
# include <stdio.h>
# include <conio.h>
int main ()
{
  int n;
for(int i=1;; i++)
{
scanf("%d", &n);
printf("The Entered number is %d \n", n);
if (n==-1)
{
printf("The program is terminated");
break;
}
getch();
return 0;
}
```

(4/Jan/2023)

(for loop)

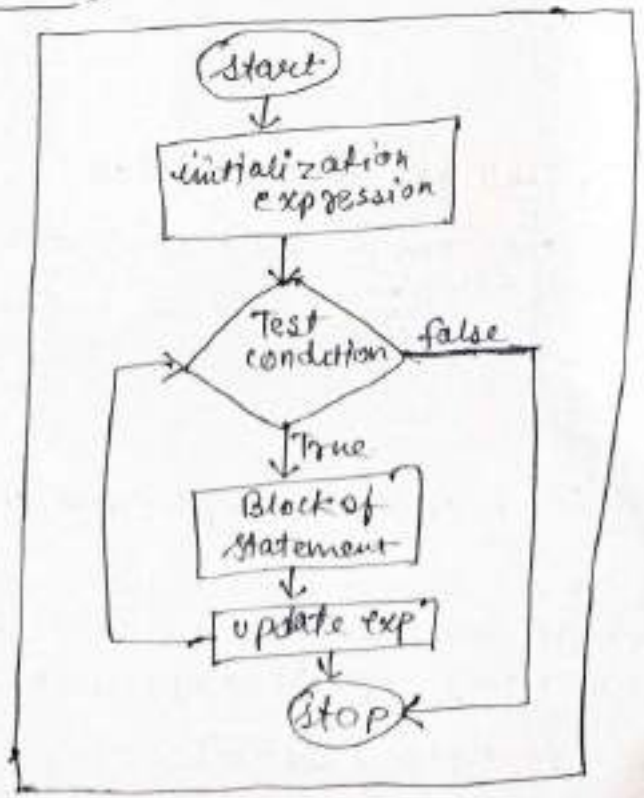
- for loop in c programming is a repetition controlling structure that allows programmers to write a loop that will be executed a specific number of times. It also executes the code until is false.
- for loop enables programmers to perform n number of steps together in a single line.
- In this 3 parameters are given :-

1) initialisation
2) condition
3) increament & decrement.

Syntax:-

```
for (initialization; condition; inc/dec)
{
    // statements;
}
```

(flowchart)



(PPS) 4/JAN/2023

Eg ①:

```
#include <stdio.h>
#include <conio.h>
int main()
{ int i;
  for(i=1; i<=10; i++)
  { printf("Hello world");
  }
  getch();
  return0;
}.
```

Main closed.

Statement closed.

Example

output screen

Hello world
Hello world
Hello world
Hello world
Hello world.

10 times like this

Eg ②:

```
#include<stdio.h>
#include< conio.h>
int main()
{ int i, number;
  for(number=1; number<=10; number++)
  { printf("%d\n", number);
  }
  getch();
  return0;
}.
```

Main closed

Statement closed

(Initialisation expression)

- In this expression we have to initialise the loop counter to some value.

∴ foreg:- (int i=1)

(Test/condition expression)

In this expression we have to test the condition if the cond^n evaluates to true then we will execute the body of loop and go to update expression else exit from loop.

foreg:- (i<=10)

(PPS) 4/Jan/2023

(update expression)

After executing the loop body this expression inc/dec the loop variable by some value.

for eg:- (i++) or (i--)

(Multiple loop Variables)
or
(Nested loop in C)

- C programming allows you to use one loop inside another loop
- Any number of loops can be define d inside another loop that is there is no restriction for defining any number of loops.
- You can define any type of loop inside another loop for example:- You can define while loop inside for loop.
- A nested loop means a loop statement inside another loop statement.
- That's why nested loop are called as loop inside loop.

(Nested for loop)

The nested for loop means any type of loop which is defined inside the for loop

* Syntax:-

```
for (initialisation; condition; inc/dec)
{ for (initialisation; cond; inc/ Dec)
  { block of statements; // inner loop statement
  }
  block of statements; // outer loop statement
}
```

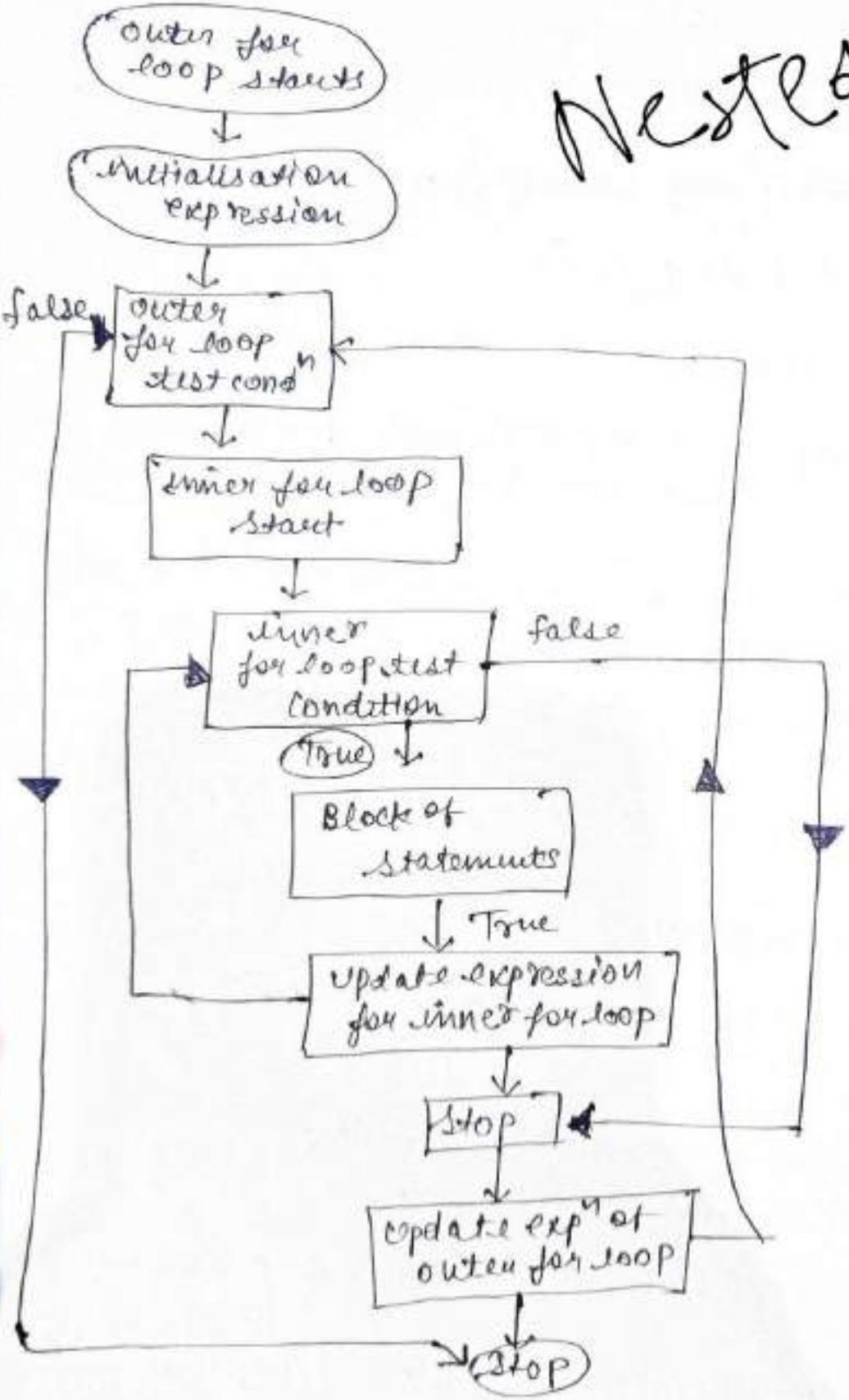
for loop closed.

Block (✓)

(PPS) 4/Jan/2023

(flowchart)

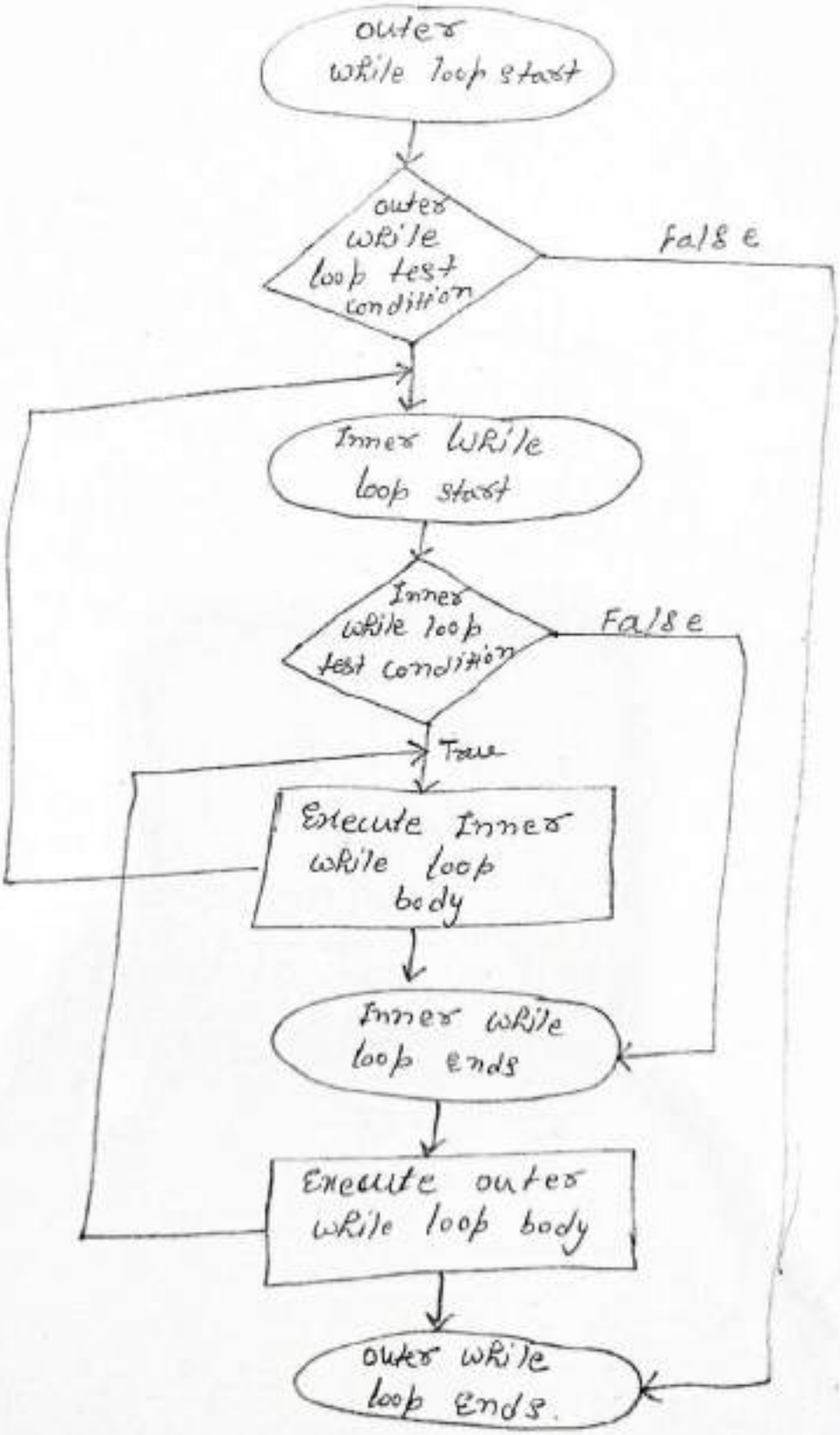
Nested for loop



PPS 4 Jan 2023

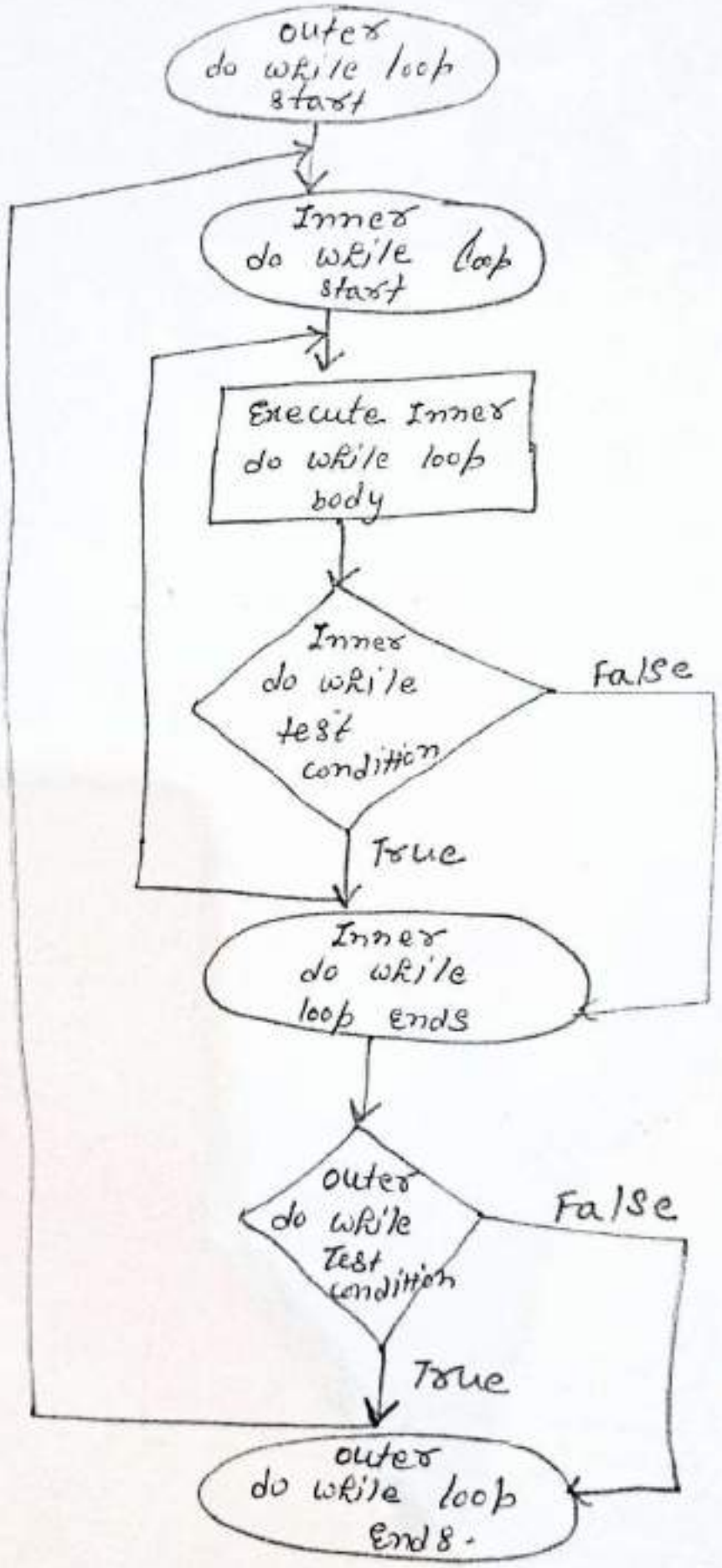
Nested while loop

Flow diagram for nested while loop.



Nested do-while loop

Flow diagram for Nested 'do-while' loops:



(PPS) 04/Jan/2023

(Nested while loop)

* (Syntax) *

```
while (outer condition)
{
while (inner condition)
{ //inner loop statements;
}
//outer loop statements;
}.
```

(Nested do while loop)

*(Syntax) *

```
do
{
do
{ //inner loop statements;
}
while (inner condition);
//outer loop statements;
}
while (outer condition);
```

(Program for nested loops)

Eg:- Nested for loop

① /* Program to print the pattern */

```
#include<stdio.h>
#include<conio.h>
int main()
{ int i, j ;
for (i=1; i<=6; i++)
{ for (j=1; j<=i; j++)
  { printf (" *")
  }printf ("\n");
  }
  getch ();
  } return 0; }.
```

o/p screen

```
*
* *
* * *
* * * *
* * * * *
* * * * * *
```

Eg:- Nested for loop

② /* Program to print the pattern */

```
#include<stdio.h>
#include<conio.h>
int main()
{ int i= 1;
  while (i<=5)
  { int j=1;
    while (j<=5)
    { printf ("%d", j);
      j++;
    }
  Printf ("\n");
  i++; }
  return 0;
  getch();
  }
```

o/p screen

```
1 2 3 4 5
1 2 3 4 5
1 2 3 4 5
1 2 3 4 5
1 2 3 4 5
```

Eg:- Nested for loop (PPS) 04/Jan/2023

/* Program to print the Pattern */

```
#include<stdio.h>
#include<conio.h>
int main()
{ int i=3, j;
  clrscr();
  do
  { j=1;
  do
   { printf(" * ");
     j++;
   }.
  while (j<=i);
  printf("\n");
   i--; }
  while (i>=1);
  getch();
  return 0;
  }.
```

O/P screen

```
* * *
* *
*
```

Break Inside Nested Loops

Whenever we use a break statement inside the nested loops it breaks the innermost loop and program control is inside the outer loop.

E.g

```
# include <stdio.h>
# include <conio.h>
int main ()
{
 int i= 0;
for(int i=0; i<5 ; i++)
 {
 for(int j=0; j<3 ; j++) // This inner loop will
                            break when i==3
 {
  if (i==3)
  {
   break ;
  }
  printf("* ");
 }
 printf("\n");
}
getch();
return 0;
}
```

Continue Inside Nested loops

whenever we use a continue statement inside the nested loops it skips the iteration of the innermost loop only. The outer loop remains unaffected.

Eg

```
# include <stdio.h>
# include <conio.h>
int main ()
{
 int i = 0;
for (int i = 0; i<5; i++)
{
for (int j = 0; j<3; j++) // This inner loop will
                             skip when j==2
{
 if (j==2)
{
continue;
}
 printf("%d", j);
}
printf("\n");
}
 getch();
 return 0;
}
```

UTKARSH PROGRAMMING SECTION 11

(Program to check for leap Year)

```
# include < stdio.h>
# include < conio.h>
int main()
{ int year;
  printf (" enter year");
  scanf ("%d", &year);
   if (year % 4 == 0)
   if ( year % 100 == 0)
   if ( year % 400 == 0)
     printf ("Yes, the year is leap year");
      else
      printf ("No, the year is not a leap year");
      else
       printf (" No this is not a leap year");
       getch();
        }.
```

(function in c) (16-Jan-2023). PPS

Programming Fundamentals for Problem Solving

(function)

* A function is a block of code which only runs when called.
* The function contains a set of programming statements enclosed by curly braces. { }.
* it has a name for identification, A function can be called multiple times provides code reusability and modularity to the C program.

* [There are the following Advantages of C functions] *

- By using functions we can avoid rewriting same logic of code again & again in a program.
- we can call C functions any function / number of time in a program and from any place in a program.
- we can track a large C program easily when its ÷ into multiple functions.
- reusability is the main Achievements of C function

* (function Aspects) *

- There are 3 aspects:-

1) function Declaration or [function prototype.] Another Name. VIMP.

= A function declared globally in a C program to tell the computer about the function name, function parameter, return type.

Syntax:-

return type function name (parameter list); or (Argument),
or
(function type)

Eg:- * function declaration is as follows.

function Parameter type.

int max(int num1, int num2); → ending statement semicolon.

return type — function — Parameter list or Parameter name.

- The Actual Body of the function can be defined seperately.
- parameter names are not important in function declaration only their type is required so the following is also the valid declaration.

Eg:- int max (int, int)

(ii) Function Defination :-

- It contains the actual statements which are to be executed.
- It is most important aspect to which the control comes when the function is called.
- Here we must notice that only one value can be returned from the function.
- If function defination in C programming consists of a function header & body.

Syntax:-

```
return type function name(Parameter list)
{
  body of the function
}
```

Eg:-

return type. return type. Parameters (arguments)

```
int max( int num1, num2)   ← No semicolon.
{ // statements
    return 0;   // a return statement.
}
```

header. Body. Local variable Declaration.

PPS 16/Jan/2023

(iii) function call:-

- To call a function parameters are passed along the function name
- function can be called from anywhere in the program.
- The parameter list must not differ in function calling and declaration
- it must pass the same number of functions as it declared in the function declaration.

Syntax:-

function name (argument list or Parameter list)

S.no	C function aspects.	Syntax.
1	function declaration.	return type function name (AL);
2	function defination	return type function name (AL)
3	function call	function name(AL)

(Types of functions)

There are 2 types of function in C programming:-

1) Library function or Built-in function or Predefined

- These are the functions which are declared in the header files of C.

eg:- scanf(), printf(), getch(), clrscr(), gets(), puts().

2) User defined functions

These are created by C programmer, so that he can use it many times. it reduces the complexity of a big program and optimizes the code.

PPS 17/JAN/2023 2022CS10T102

Examples:

```
#include <stdio.h>
#include < conio.h>
void main ()
{ void is even(),
  void add ();
   closer(),
   add();
   is even(),
   getch(),
}
```

// func^n declaration.

function called.

predefined function.

```
void add()
{ int a,b,c;
  printf ("Enter two no");
  scanf ( "%d %d", &a, &b);
   c = a+b;
  printf (" sum is %d", c)
}
```

```
is even()
{ int a;
  printf(" Enter a number");
  scanf ("%d", &a);
     if (a%2 == 0)
     Printf (" even")
     else
     printf("odd");
   }.
```

ONLY FOR YOUR UNDERSTANDING

Function Declare

int add(int a, int b)
int add (int, int);

function defination

int (int x, int y).

PIS 17/Jan/2023 2022CS10702

Example: #
#
for multiple function calls:

```
main()
{
  a();
  b();
  a();
}
a()
{ printf("hello");
}
b()
{ a(); }
```

#(Return Value)#

- A function may or may not return a value from the function.
  2) A C function may not want to return any value from the function use void return type

#(Example without return value)#

```
void hello()
{ printf("hello");
}
```

- If you want to return any value from the function you need to use any data type such as int, float, long, char etc.
- The return type depends on the value to be returned from the function.

e.g:- (with return value)

```
int get()
{ return 10; }
```

- In the above example we have to return 10, as a value, so the return type is int.
- If you want to return floating points value (example 10.2, 3.1, 54.5 etc) you need to use float as the return type of the method.

Eg :- With return value

```
float get()
{ return 10.2;
}
```

(Different Aspects of function calling)

- A function may or may not accept any arguments.
- It may or may not return any value.
- Based on these facts there are 4 different Aspects of function calling.

(18/Jan/2023)

(i) Function Without Args() & without return Val().

OR

No Args() & No return value.

```
# include <stdio.h>
# include < conio.h>
    void sum();   // function Declaration
    void main().
    {
     clrscr();
     sum();      // function Calling.
     getch();
    }
    void sum()   // function defination.
     {int a,b,c;
     printf(" Enter two numbers");
     scanf("%d %d", &a, &b);
        c = a+b;
     printf(" Addition %d", c);
     }.
```

PPS 18/Jan/2023 2022CS10702

(ii) function without Args(), With return value()
OR
No Args() & Return value.

```
#include <stdio.h>
#include<conio.h>
void main()
int sum();      // function Declaration.
{ int s;
  clrscr();
  s = sum();    // function calling.
Printf ("Addition %d", s);
getch();
}
int sum()       // function defination.
{ int a,b,c;
  printf(" Enter 2 numbers");
  scanf ("%d %d", &a,&b);
    c=a+b;
    return c;
}
```

(iii) function with Arguments(), without return val().
OR
No return value but Args()

```
#include<stdio.h>
#include<conio.h>
void sum(int x, int y);  // function Declaration.
int main()
{ int a,b;
  Printf ("enter two no"");
  scanf ("%d%d", &a,&b);
  sum(a,b);  // function calling.
}
```

```
void sum(int x, int y)   // function defination
{
  int z;
   z = x+y;
  Printf ("sum of two no is %d", z);
}
```

Note
when we don't return the value we printf the o/p inside a function.

(iv) <u>function with Args () & with return Value ()</u>

```
#include<stdio.h>
#include<conio.h>
int sum (int x, int y);  // function declaration
                            or function Prototype
  main()
{int a, b, c;
 printf ("Enter two no""");
 scanf ("%d %d", &a, &b);
   c = sum (a, b)      // function calling
   printf ("sum of 2 no's is %d", c);
}
int sum (int x, int y)   // function defination.
{ int z;
  z = x+y;
   return (z);
}
```

IMPORTANT TOPICS (H)

- Aspects (V)
- functions
- Call by Value & Ref (Sureshot)

(19/Jan/2023) # (Passing Parameters through functions) # (80)
or (function Arguments)

There are two ways of passing parameter to the function:-

① <u>Actual Parameter</u>:- ⊙ these are the values, variables or expressions that we pass through a calling function.

⊙ while defining the actual parameters in the calling functions we don't associate data type with the parameter.

⊙ In a calling function the actual parameter can be provided

either by coder at time of coding or by the user in real time, when the program is being executed.

Exp ① *In the example below the actual parameter are passed by the user of the program in real time.

```
#include <stdio.h>
#include <conio.h>
int add (int x, int y)   → function Declaration
                           // x & y are formal parameter.
{int sum;
 sum = x+y;
 return sum;
}
void main()
{ int a, b, result;
 printf("Enter 2 value");
 scanf("%d %d", &a, &b);
 result = add(a,b);   /* function calling
                          a & b are Actual parameters. */
 Printf("result of addition is %d", result);
   getch();
}
```

---

*In the Program below the Actual parameters are passed by the coder by coding.

Example ②

```
#include <stdio.h>
#include <conio.h>
int add (int x, int y)   // x & y formal parameter.
{int sum;
 sum = x+y;
 return sum;  } void main().
{result = add (23, 42);   // 23 & 42 are Actuals.
Printf ("result of addition is %d", result);
 getch();
}.
```

Formal Parameters

- formal parameters are the variables that we pass to the function when its declared & defined in program.
- The formal parameters defined during function defination. accepts the real values in the order they are passed by the calling func.
- The statements in the func^n Defination Accesses these real values by referring with the variable name defined by the formal parameter.
- formal paras are always passed to a function defination along with their associated datatype.

"we use Actual parameters while calling a function". while we use formals for declaring & defining a function.

IMPORTANT

(while calling a function there are two ways in which args () can be passed to a function)

(i) Call by Value. (values are Passed)

- int x = 10, y = 20; // func^n Declared
  fun (x, y);

Printf ("x = %d, y = %d", x, y);

* Pointer: holds the value of Address. other variable,
Denoted by (*),
Dereference operators
(Access the content in the variable using its Address).

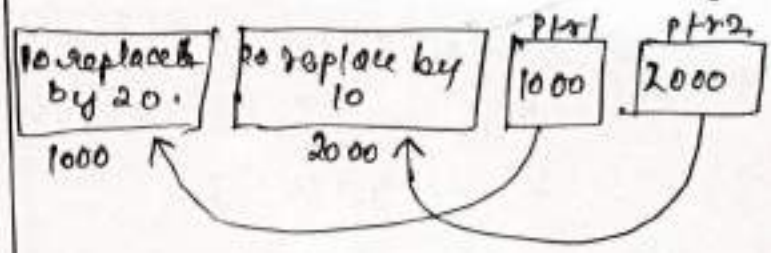
(ii) Call by reference (Addresses Passed)

- int x = 10, y = 20; // func^n Declaration.
  fun (&x, &y); // calling.
  ↳ Address of operator or reference operator.

Address is passed not values.

int fun (int *ptr1, int* ptr2)
↓ way to declare Pointers.

{ *ptr1 = 20;
*ptr2 = 10;
}. ← {Access the value that particular address.}.



- printf ("x = %d, y = %d", x, y);

Copies memory location at particular Address.

* Dereference Operator.

* Key Differences between Actual and Formal Parameters.

1) Actual Parameters are the genuine values that are passed on to the called function.
However, the formal parameter is the variable defined to receive the values when the corresponding function is invoked.

2) Actual parameters are one that occurs when we call a function.
Formal parameters occur when we declare or define a function.

3) The actual parameter can be passed either by the programmer itself or by the user in real time.
The Formal Parameter is always defined by the programmer.

4) We do not define data type along with the actual parameter whereas, we define data types along with the formal parameter.

5) The Actual Parameter can either be a value or a variable.
On the other hand, the formal parameter is always a variable.

6) The actual Parameters are defined as: add (a, b);

7) The formal Parameters are defined as:

```
int add (int x, int y) {
//code
```

Conclusion

Actual and formal parameters are two different forms of parameters. we pass actual parameters to a calling function whereas, we pass formal parameters while defining or declaring a function.

While passing actual parameters we need not mention data types. However, passing formal parameters we have to mandatorily mention their corresponding data types.

Call by value & Call by reference

*

① In the call by value method original value is not modified.	① In call by reference method, the original value is modified.
② In call by value, a copy of the variable is passed.	② In call by reference, a variable itself is passed.
③ In Call by value method the value of the variable is passed to the function as parameter.	③ In call by reference method the address of variable is passed to the function as parameter.
④ The value of actual parameter can not be modified by formal parameter.	④ The value of the actual parameter can be modified by formal parameter.
⑤ Different memory is allocated for both actual and formal parameters because value of actual parameter is copied to formal parameter.	⑤ Same memory is used for both actual and formal parameters since only address is used by both parameters.
⑥ Any update made inside function will not affect the original value of variable in calling function.	⑥ update made inside function may affect the original value of variable in calling functions.

Eg. Swap the value of two variables using call by value.

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

void main()
{
  void swap (int, int); // Funⁿ declaration
   int a, b;
 printf(" Enter the value of a, b");
 scanf ("%d %d", &a, &b);
 swap (a, b); // Funⁿ call (Passing value to funⁿ)
}

void swap (int a, int b) // Funⁿ defination
{
   int temp;
    temp = a;
     a = b;
     b = temp;
 printf(" After swap a = %d b = %d", a, b);
  getch();
}
```

O/P Enter the value of a, b

100, 200

a = 200

b = 100

Eg Swap the value of two variables using call by reference.

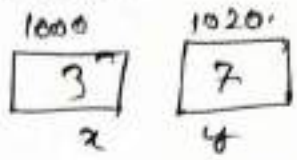
```
# include <stdio.h>
# include <conio.h>
void main ()
{
  int a,b;
Void swap (int* , int*); // Funⁿ declaration
printf (" Enter the value of a,b");
Scanf ("%d %d", &a, &b);
 Swap (&a, &b); // Funⁿ call (Passing value to function)
}

void swap (int *a , int *b) // Funⁿ defination
{
   int temp;
   temp = *a;
   *a = *b;
   *b = temp;
   printf (" After swap a=%d, b=%d", *a, *b);
}
```

(20JAN2023) PPS

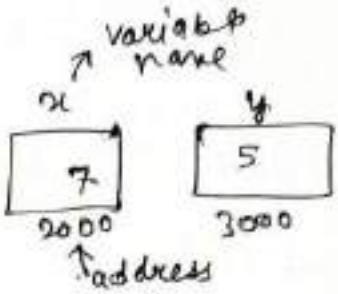
Call by value

```
# include < stdio.h >
# include < conio.h >
void fun (int, int);   // funⁿ declaration.
void main ();
{
int x=5, y=7;
clrscr();
fun (x,y);   // funcⁿ calling.
```

Actual Argument



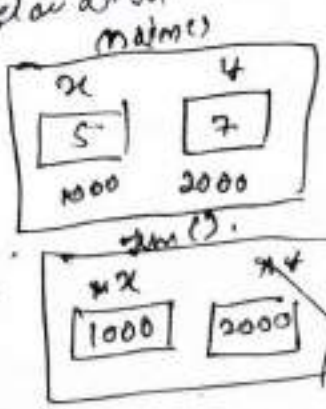

```
printf ("x = %d, y = %d", x,y);
getch();
}
void fun (int x, int y)
{ x=7; y=5;
printf ("x = %d, y = %d", x,y);
}
```



Call by Reference

```
# include < stdio.h >
# include < conio.h >
void fun (int *, int *);   // function Declaration
void main()
{ int x=5, y=7;
fun (&x, &y);
printf ("x = %d, y = %d", x,y);
getch();
}
void main fun (int *x, int *y)
{ *x=y; *y=5;
printf ("x = %d, y = %d", *x, *y); }.
```

Pointer variable.



dereference operator

o/p = x = 7, y = 5
x = 7, y = 5

(Recursion):-

```
void add();   — function prototype
main()
{
add();        — function calling
}.
void add      → function definition
{
}
```

(Normal method of calling)

IMP ASKED QUESTS.
prog:- WAP print factorial.

```
void add()
{
add();   // function calling.
}.
```

(Recursive Program)

some functions calling itself.

23/Jan/23

Recursion:-

```
void add();   — Declaring
main()
{ add();      — Calling.
}
void add()    — Defining.
{ = }
```

- Recursion means calling the function itself.
- when the condition is satisfied we return back where the function was called.

Eg:-

```
#include <stdio.h>
#include <conio.h>
int main()
{
int k;
k = f1(3);        → f1 func^n call & pass 3
printf("%d", k);
}
int f1(int n)     //function definition
{ int s;
  if (n==1)
    return 1;
  s = n+f1(n-1);  return s;
}
```

Main function

Pass args()

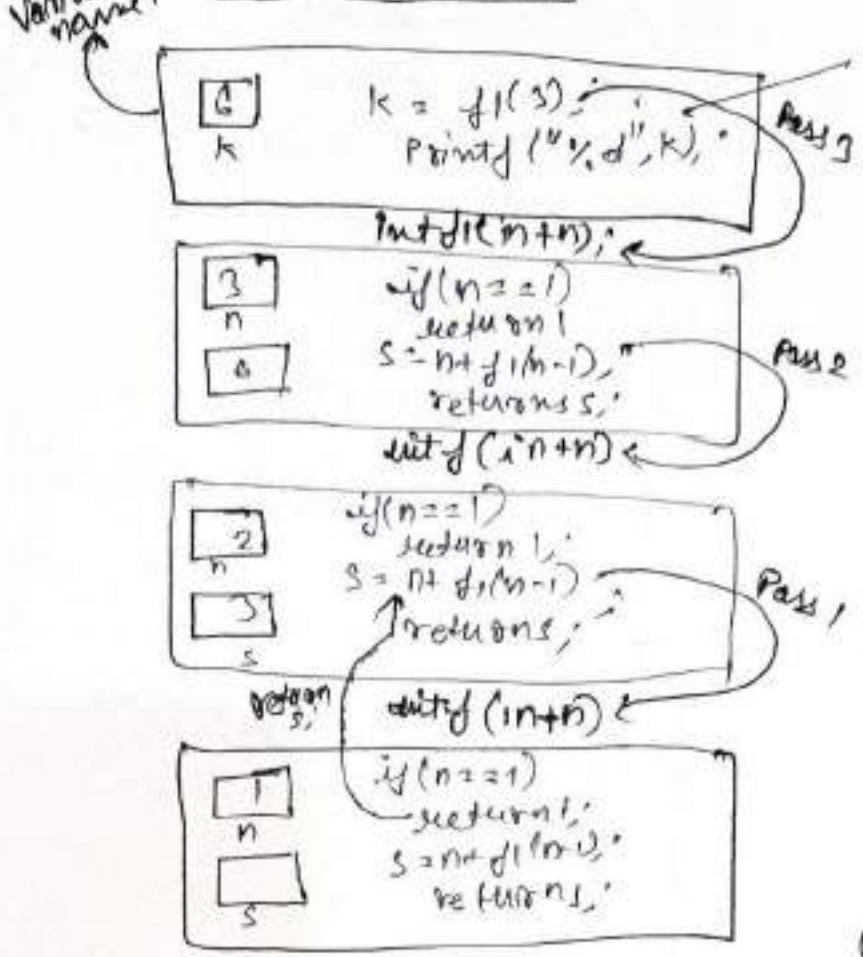
function name.

Recursive function.

PPS (24/Jan/2023)

2 marks

Execution Process



(24/Jan/2023)

* Storage Class *

storage class in c are used to determine the lifetime, visibility, memory creation and initial value of a variable there are four types of storage class in C.

① Automatic ② Register ③ Static ④ External.

Storage class	Keyword	Default Value	storage
① Automatic	Auto	Garbage	RAM
② Register	register	Garbage	CPU register.
③ Static	Static	0	RAM
④ External	Extern	0	RAM
int x; Printf ();			

PPS

Scope	Lifetime
• limited to the block in which it is declared. (within block) ↓	• Till the execution of the block in which it is declared (end of block) ↓
• same ↓	same
• same (within block)	• Till the end of program.
• Global	• Till the end of program

(Automatic)

Eg.1)
```
# include< stdio.h>
# include < conio.h >
void main()
{ auto int a =20;
  Printf ("%d", a); }.
```
↳ a = 20

Eg.2)
```
# include<stdio.h>
# include < conio.h>
void main()
{ auto int a = 10;
  { auto int a;
    Printf("%d", a);
  } printf ("%d", a);
}
```

(Register)

```
# include< stdio.h>
# include conio.h>
void main()
{ register int a=10; //fast Acess time
  clrscr();
  Printf("%d", a);
  getch(); }
```

(External)

```
# include< stdio.h>
# include < conio.h>
int a=7;
void main()
{ extern int a;
  Printf("%d", a); }.
```

Static

```
# include < stdio.h>
# include < conio.h>
void fun();
void main()
{ fun();
  fun();
  fun(); }
```
} func^n calling

```
Void fun()        // fun^n defination.
{ int a=1;        // total variable.
  static int b =1;
  Printf ("%d %d", a,b);
  a++, b++; }
```

a = 1, 1, 1

b = 1, 2, 3

Recursive Function

* A recursive function performs the tasks by dividing it into the subtasks.
* There is a termination condition defined in the function which is satisfied by some specific subtask.
* After this, Recursion stops and the final result is returned from the function.
* The case at which the function doesn't recall is called the base case whereas the instances where the function keeps calling itself to perform a subtask, is called the recursive case.

Example 1 Sum of Natural Numbers using Recursion

```
#include <stdio.h>
#include <conio.h>
int sum(int n);
int main ()
{
    int number, result;
  printf("Enter a positive integer:");
  scanf("%d", & number);
  result = sum(number); // function call
  printf("sum = %d", result);
  return 0;
}
int sum(int n)
{
  if (n != 0)
  return n + sum(n-1); // sum() function calls itself
  else
    return n;
}
```

Passed as an argumen

Output

Enter a positive integer : 3

sum = 6

Execution Process

```
# include <stdio.h>
# include <conio.h>
int main ()
{
  . . . . . .
  result = sum(number);
  . . . . . .
}
```

```
int sum (int n)
{
  if (n != 0)
    return n/3 + sum(n-1)
  Else
    return n;
}
```

3+3 = 6 is returned

```
int sum (int n)
{
  if (n != 0)
    return n/2 + sum(n-1)
  Else
    return n;
}
```

2+1 = 3 is returned

```
int sum (int n)
{
  if (n != 0)
    return n + sum(n-1)
  Else
    return n;
}
```

1+0 = 1 is returned

```
int sum (int n)
{
  if (n != 0)
    return + sum(n-1)
  Else
    return n;
}
```

0 is returned

Imp

E.92 Factorial of a number using Recursion.

```
#include <stdio.h>
#include <conio.h>
int fact (int);
int main ()
{
int n, f;
printf (" Enter the number whose factorial you
        want to calculate");
scanf ("%d", &n);
f = fact (n);
printf ("factorial = %d", f);
getch ();
}
Int fact (int n)
{
if (n==0)
{
return 0;
}
Else if (n==1)
{
return 1;
}
Else
{
return n*fact (n-1);
}
}
```

O/P

Enter the number whose factorial you want to calculate 5

factorial = 120

Imp E.93 Fibon

Recurs

```
# include
# include
int fib
{
if (i
{
retu
}
if (
{
re
}
ret
}
int
{
in
for
{
pr
}
r
}
```

Prog 3 Fibonacci series for a given number using a Recursion.

```
# include <stdio.h>
# include <conio.h>
int fibonacci (int i)
{
  if (i==0)
  {
  return 0;
  }
  if (i==1)
  {
  return 1;
  }
  return fibonacci (i-1)+ fibonacci (i-2);
}
int main ()
{
  int i;
  for (i=0; i<10; i++)
  {
  printf("%d \n", fibonacci (i));
  }
  return 0;
  getch ();
}
```

O/P

0
1
1
2
3
5
8
13
21
34

Storage classes in C

Automatic

* Automatic variables are allocated memory automatically at runtime.
* The keyword used for defining automatic variables is auto.
* Hence, the keyword auto is rarely used while writing programs in C language.
* The scope of the automatic variables is limited to the block in which they are defined.
* Every local variable is automatic in C by default.
* They are assigned a garbage value by default whenever they are declared.

② Register

* The register storage class is used to define local variables that should be stored in a register instead of RAM
* This makes the use of register variables to be much faster than that of the variables stored in the memory during the runtime of the program.
* The register keyword is used for the variable which should be stored in the CPU register.

* The access time of the register variables is faster than the automatic variables.

* We cannot store pointers into the register; i.e. a register can store the address of a variable.

* Static variable can not stored into the register since we can not use more than one storage specifier for the same variable.

Static

* The static storage class instructs the compiler to keep a local variable in existence during the life-time of the program instead of creating and destroying it each time it comes into and goes out of scope.

* Therefore, making local variables static allows them to maintain their values between function calls.

* Static variable have the property of preserving their value even after they are out of their scope! Hence, static variables preserve the value of their last use in their scope.

* The keyword used to define static variable is Static

(4) External

* Extern storage class simply tells us that the variable is defined elsewhere and not within the same block where it is used.
* The variables declared as extern are not allocated any memory.
* An external variable can be declared many times but can be initialized at only once.
* We can only initialize the extern variable globally, i.e. we can not initialize the external variable within any block or method.
* The extern storage class is used to give a reference of a global variable that is visible to all the program files.

Example of External

eg1

```
#
#
int a;
int main()
{
Extern int a; // variable a is defined globally, the memory will not be allocated to a
printf("%d", a);
}
```

eg2

```
#
#
int a;
int main()
{
Extern int a=0; // this will show a compiler Error since we can not use extern and initializer at same time.
printf("%d", a);
}
```

eg3

```
#
#
int main()
{
Extern int a; // compiler will search here for a variable a defined and initialized somewhere in the program or not.
printf("%d", a);
}
int a = 20;
```

o/p → 20

Eg4

```
Extern int a;
int a = 10;
#include <stdio.h>
#include <conio.h>
int main()
{
printf("%d", a);
}
```

int a = 20; // compiler will show an Error at this line because variable can be initialized at only once in External storage class.

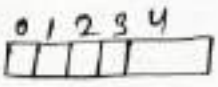
U TKARSH
CDT102
AGGARWAL

(Array)

25th Jan 2023 PPS

- An array is defined as the collection of similar type of data items stored at continuous memory location.
- Arrays are the derived data type in c-programming language which can store the primitive type of data such int, char, double, float etc.
- it all has the capability to store the connection of derived data type such as pointers, structure etc.
- The array is the simplest data structure where each data element can be randomly accessed by using its index number.
- c array is benefit if you have to stored similar element.
- By using array we can access the element easily.
- Array elements are always counted from 0 onwards.

* eg: arr [0] → index of the element to be accessed.
  array variable area



* eg:

40	55	65	75	80	90	10
0	1	2	3	4	5	6

25/Jan/2023 PPS

(Properties of Array)

Array contain following properties.

① Each element of an array is of same data type and carries the same size that is int = 4 bit.

② element of the array stored as it in memory-location where as the 1st element is stored at the smallest memory.

③ Element of array can be randomely accessed since we can calculate the address of each in array with the given base.

④ We can declare an array in the C language in following way.

(Declaration of array)

data type array name[array size]

Eg = int mark[5];

(Initialization of Array)

The simplest way to initialize an array is by using the index of each element.

```
Eg: marks[0] = 80;   // initialization of array.
    marks[1] = 60;
    marks[2] = 90;
    marks[1]
```

80	60	90	
Mark[0]			

```
int num=10;  int num=20;  int num=30;  int num=40;
int number[4] = { };
```

10	20	30	40
0	1	2	3

To call data by index value;

```
Printf ("%d", number[0]);
Printf (      );
```

Eg ①

```
#include<stdio.h>
# include<conio.h>
int main()
{int i=0;
 int marks[5];      // declaration of Array.
 marks[0]=80;       // initialising of array.
 marks[1]=60;
 marks[2]=70;
 marks[3]=85;
 marks[4]=75;
for(i=0; i<5; i++)  // traversal of array.
{
printf("%d\n", marks[i]);
 }                    → 0
getch(); }.
```

Eg ②

```
#include<stdio.h>
# include<conio.h>
int main()
{int i=0;
int marks[5]={20, 30, 40, 50, 60}; // declaration
```

(Array Declaration)

we can initialise in C array at the time of declaration.

int marks[5] = {20, 30, 40, 50, 60};

in such case there is no requirement to define the size.

int marks[] = {20, 30, 40, 50, 60};

(Program of user input in Array)

PPS 27/Jan/2023

```
# include<stdio.h>
# include<conio.h>
# define size 5
```

→ used only when we need to get input by user

```
void main()
{ int i;   // used for loop.
  clrscr();
int studmarks[size];  // Array Declaration.
for (i=0; i<size; i++)
{ printf("enter the marks of %d subject =", i+1);
  scanf ("%d", &studmarks[i]);
}
```

→ value of i

→ store value at index[0].

```
printf ("\n Data you entered");
  for(i=0; i<size; i++)
  { printf("\n value in index [%d] is %d", i,
          studmarks[i]);
    getch(); }.
```

O/P:-

```
Enter the marks of subject = 2
      "        2          = 3
      "        3          = 4
      "        4          = 5
      "        5          = 6
```

```
PA" Data You Entered).
Value of index [0] is 2
      "        [1] is 3
      "        [2] is 4
      "        [3] is 5
      "        [4] is 6
```

VIMP #(C Program to count Even & odd elements and display it in screen) #

Sureshot

Ans:-

```
# include< stdio.h>
# include< conio.h>
# define N 10
  int main ()
{
int a [N] i, e, even=0, odd=0
```

PPS 27/Jan/2023

```
printf("Enter %d integer numbers \n", N);
for(i=0; i<N; i++)
scanf("%d", &a[i]);
printf("\n Even numbers in array are \n");
for(i=0; i<N; i++)
{ if (a[i]%2 == 0)
  { printf("%d \t", a[i]);
    even++;
  }
}
printf("\n odd numbers in the array are \n");
for(i=0; i<N; i++)
{ if (a[i]%2 != 0)
  { printf("%d \t", a[i]);
    odd++;
  } }.

printf("\n total even no: %d", even);
printf("\n total odd no: %d", odd);
getch();
}.
```

:(Output):-

```
Enter 10 integers.
1
2
3
4
5
6
7
8
9
10
```

Even no in Array = 2 4 6 8 10
odd no in Array = 1,3 5 7 9
Total even no = 5
Total odd no = 5.

PPS 30 Jan 2023

Q1:

a $\begin{bmatrix} 0 & 5 & 2 \\ 1 & 1 & 0 \\ 7 & 4 & 3 \end{bmatrix}$ b $\begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 3 & 5 \end{bmatrix}$ = c [

```
for ( i=0; i<=2; i++)
{
for (j=0; j<=2; j++)
 { sum=0
 for (k=0, k<=2; k++);
  { sum = sum + a[i][k] * b[k][j];
   }
   c[i][j] = sum; }
```

(Advantages of array)

(i) code optimization :-

a) less code to access data.

b) Easy of diversing :-
by using the for loop we can retrive the data in an array.

c) easy of sorting :- to sort an array we need a lines of code only

d) Random Access :- we can access any element randomly using an array.

(Program to print matrix) # 31/Jan/2023

```
# include <stdio.h>
# include <conio.h>
void main()
{ int a[3][3], i, j;
clrscr();
for (i=0; i<3; i++)  ⟶ 1st row.
{
for (j=0; j<3; j++)  ⟶ 1st column.
{ printf("value for [%d][%d]", i, j);
scanf("%d", &a[i][j]);
}
}
printf("\n matrix");
for (i=0; i<3; i++)
{ for (j=0; j<3; j++)
{ printf("%d\t", a[i][j]);
}
printf("\n");
}
getch(); }
```

(C Program to calculate sum of odd & even Num) # in Array.

Sure Shot

Ans

```
# include < stdio.h>
# include < conio.h>
  void main()
   {int a[10], i, S1 = 0, S2 = 0;
printf (" Enter 10 numbers for array");
  for (i=0; i<=9; i++)
   {scanf ("%d", &a [i]);
    }
   for (i=0; i<=9; i++)
  { if (a[i] % 2 == 0)
      S1 = S1 + a[i];
    else
      S2 = S2+a[i];
   }
 printf (" \n sum of even =%d", S1);
 printf (" \n sum of odd =%d", S2);
  getch();
     }.
```

VVIMP

(WAP to multiply matrix)

```
# include< stdio.h>
# include <conio.h>
  void main()
 { int a[3][3], b[3][3], c[3][3] i, j, k, sum.
 printf (" Enter no for 1st Matrix"),
   for ( i=0; i<=2; i++)
   for (j=0; j<=2; j++)
 Scanf ("%d", &a [i][j]);
   printf ("Enter no for 2nd Matrix"),
 for(i=0; i<=2; i++)
   for (j=0; j<=2; j++)
   scanf ("%d", &b[i][j])
```

```
for (i=0; i<=2; i++)        //for multiplication.
  for (j=0; j<=2; j++)
  { sum=0;
   for(k=0; k<=2; k++)
   { sum = sum + a[i][k] * b[k][j];
     c[i][j]=sum; }
  Printf ("\n The result is");
   for (i=0; i<=2; i++)
    { for (j=0; j<=2; j++)
    {
     Printf (" %d \t", c[i][j]
     }
  Printf ("\n"); }
   getch();
     return 0; }
```

01/feb/2023 * (PROGRAM TO ADD TWO Matrix)

```
#include <stdio.h>
# include< conio.h>
  void main()
  { clrscr();
int a[3][3], b[3][3], c[3][3],i, j;
Printf ("Enter the no for 1st Matrix");
  for (i=0; i<3; i++)
  { for(j=0; j<3; j++)
  { printf ("value at index[%d][%d] ", i, j);
   scanf ("%d", &a[i][j]); } }
   Printf (" Enter the no for 2nd Matrix");
  for (i=0; i<3; i++)
   { for (j=0; i<3; i++)
     { printf ("value at index [%d][%d]; i,j);
       scanf ("%d", &b[i][j]);
       } }
```

i = row (for)
j = column (for)

i goes once in the while
j goes twice.

```
for (i=0; i<3; i++)
{ for (j=0; j<3; j++)
  { c[i][j] = a[i][j] + b[i][j]
  }
}
for (i=0; i<3; i++)        → for printing Result.
{ for (j=0; j<3; j++)
  { printf("%d\t", c[i][j]);
  }
  printf("\n");
} getch();
```

* (Program To find Transpose of Matrix) *

```
#include<stdio.h>
#include<conio.h>
int main()
{ clrscr()
  int a[50][50], b[50][50], c[50][50], i, j, m, n;
  clrscr();
printf("Enter number for row \n");
  scanf("%d", &m);
  printf("Enter number for column\n");
  scanf("%d", &n);
printf("Enter matrix element");
  for (i=0; i<m; i++)
  {
  scanf("%d", &a[i][j]);
  } }
```

So extra space Generation

```
Printf ("\n Matrix")
  for (i=0; i<m; i++)
 {
  for (j=0; j<n; j++)
   {
   Printf ("%d\t", a[i][j]);
    }
    Printf ("\n");
    }
 Printf (" transpose is");
   for (i=0; i<m; i++)
    {
   for (j=0; j<n; j++)
    {printf ("%d\t", a[j][i])
    }
    printf ("\n");
    }
    getch();
      }.
```

<u>output:-</u>

>> enter no of rows. 3

>> enter no of colums = 3

>> Enter elements: 1 2 3 4 5 6 7 8 9

>> Matrix is $= \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$

>> Transpose is $= \begin{bmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{bmatrix}$

(character Array and string)#

- In c programming string is a 1D array of characters and is defined as an array of characters.
- But an array of string in c is a 2D array of character types.
- Each string is terminated with a null character ('\0').
- A string is a collection of characters hence to define a string we use a character array
- The termination or null character ('\0') is important in a string since it is only way to identify where the string ends.
- string are the group of characters, digits and symbols, inclused in ("") marks.

C Program to find length of string

```
#
#
int main()
{
char a[50];
int i;
printf("\n Enter your name");
gets(a);
i = 0;
while (a[i] != '\0')
i++;
printf("\n Total Number of character = %d", i);
return 0;
getch();
}
```

	0	1	2	3	4	5	6
a	R	a	m	e	s	h	\0

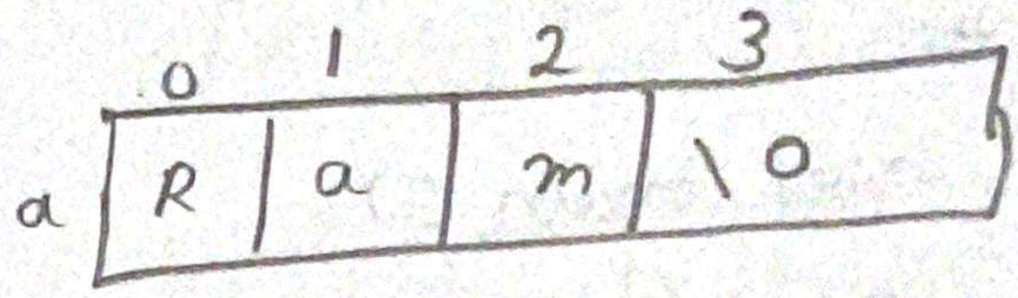
O/P

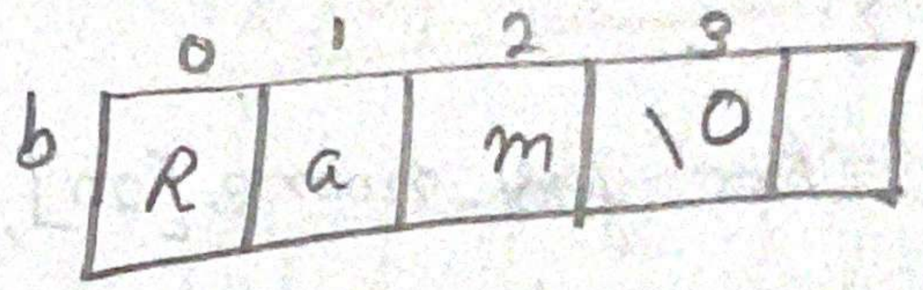
Enter your name

Total Number of character = 6

C Program to copy a string

```
#
#
#
int main ()
{
Char a[50], b[50];
int j;
i = 0;
printf ("\n Enter first string:");
gets (a);
while (a[i] != '\0')
{
b[i] = a[i];
i++;
}
b[i] = '\0';
printf ("\n copied string = %s", b);
getch ();
return 0;
}
```

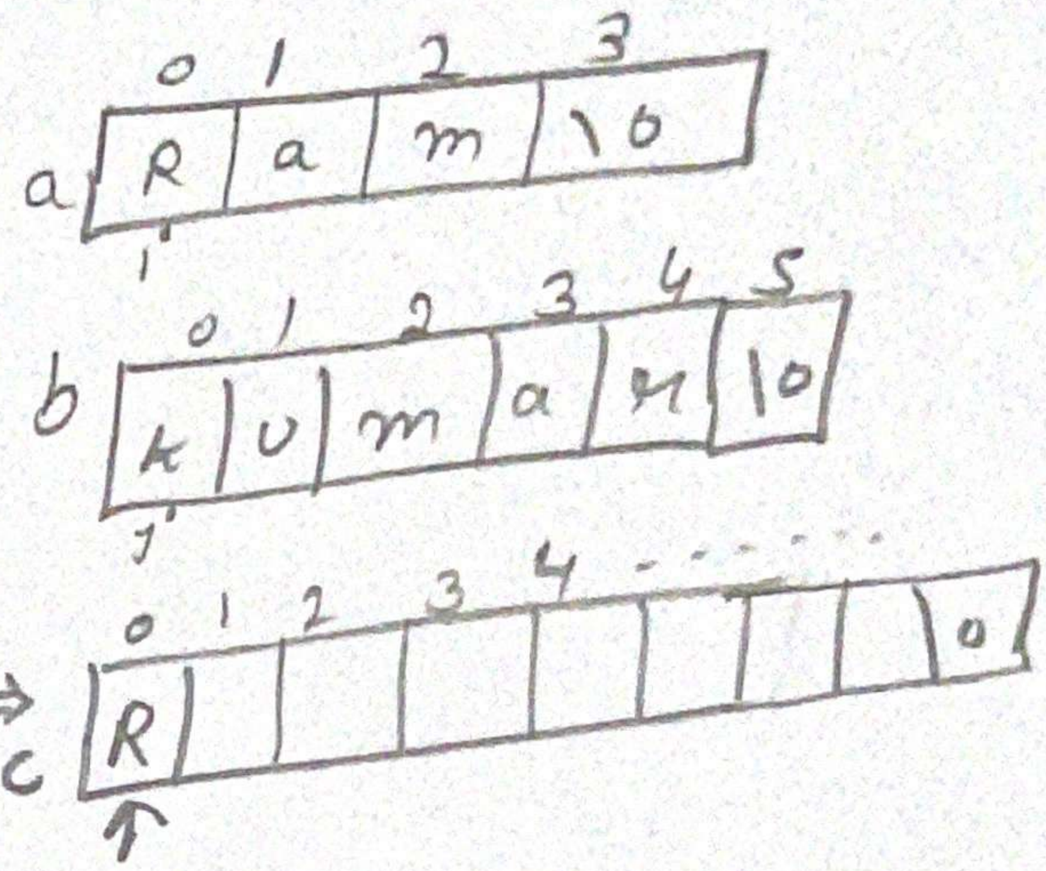




O/P

```
Enter string: Ram
Copied string = Ram
```

C Program to Concatenate




```
#
#
#
int main ()
{
Char a[50], b[50], c[ ];
int i, j;
printf("\n Enter first string");
gets (a);
printf("\n Enter second string");
gets (b);
i = j = 0; {both set as 0}
while (a[i] != '\0')
{
c[i] = a[i]; // a[R] → c[0]
                a[i]   c[i]
i++;
}
while (b[j] != '\0')
{
c[i] = b[j]; // Here c[i] means c[3]
i++; j++;
}
c[i] = '\0';
printf("\n concatenated String = %s", c);
getch ();
return 0;
}
```

Compare string without strcmp ().

```
#
#
#
Void main ()
{
Char str1[100], str2[100];
int i= 0, len1 = 0, len2 =0, same = 0;
Clrscr();
printf ("Enter first string");
gets (str1);
printf ("Enter second string");
gets (str2);
len1 = strlen(str1);
len2 = strlen(str2);
if (len1 = len2)
{
while (i< len1)
{
if (str1[i] == str2[i])
{
i++;
}
Else
{
break;
}
}
if (i== len1)
{
same = 1;
printf ("the two strings are same");
}
```

```
if (len1 != len2)
{
printf("\n The string are not equal");
}
if (same == 0)
  {
  if (str1[i] > str2[i])
   {
   printf("string 1 is greater than string 2");
   }
  Else if (str1[i] < str2[i])
   {
   printf("string 2 is greater then string 1");
   }
  getch();
}
```

```
for (i=0; i<3; i++)
{ for (j=0; j<3; j++)
  { c[i][j] = a[i][j] + b[i][j]
  }
}
for (i=0; i<3; i++)   → for printing result.
{ for (j=0; j<3; j++)
  { printf("%d \t", c[i][j]);
  }
  printf("\n");
} getch();
```

* (Program To find Transpose of Matrix) *

```
#include<stdio.h>
#include<conio.h>
int main()
{ clrscr()
  int a[50][50], b[50][50], c[50][50], i, j, m, n;
  clrscr();
printf("Enter number for row \n");
  scanf("%d", &m);
  printf("Enter number for column\n");
  scanf("%d", &n);
printf("Enter matrix element");
  for (i=0; i<m; i++)
  {
  scanf("%d", &a[i][j]);
  } }
```

→ So extra space generation

```
Printf ("\n Matrix")
  for (i=0; i<m; i++)
 {
  for (j=0; j<n; j++)
   {
   Printf ("%d\t", a[i][j]);
    }
    Printf ("\n");
    }
 Printf (" Transpose is");
   for (i=0; i<m; i++)
    {
   for (j=0; j<n; j++)
    {printf ("%d\t", a[j][i])
     }
     printf ("\n");
     }
     getch();
       }.
```

output:-

>> enter no of rows. 3

>> enter no of colums = 3

>> enter elements: 1 2 3 4 5 6 7 8 9

>> Matrix is $= \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$

>> Transpose is $= \begin{bmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{bmatrix}$

(initialization of string)

There are 4 ways to initialise a string in C.

(i) <u>Assigning a string literal without size</u>

char[] = "abcd";

- String litterals can be assigned without size, hence the name of the string 'str' acts as if pointer because its an array

(ii) Assigning a string literal with a predefined size

- we always account for an extra space which will be assigned to the null character. char str[00] =

(iii) Assigning character by character by size.

$\boxed{\text{char str[16]} = \{'s','t','r','i',\ldots\ldots '\backslash 0'\};}$

we can assign a string character by character also

(iv) Assigning character by character without size.

$\boxed{\text{char str[ ]} = \{'s','t','r','i',\ldots\ldots '\backslash 0'\};}$

The size of string is determined by the compiler. now we want to create an array of strings which means we are trying to create an array of character array

There are 2 ways to create this:

① using 2D arrays — memory gets wasted

② using pointers. — memory not waste.

① Using 2D Arrays

- Creating an string array is one of application of 2D array suppose we want to create an array of 3 string of size 5.

Eg:

B	I	a	c	k	\0
B	I	a	c	k	\0
B	I	a	c	k	\0

Null character (Represents end of string)

- Every string in a string array must terminate with a null character. its the property of an string in C language

(Disadvantage of using 2D Arrays)

- suppose we want to store 4 string in an array!

$\boxed{\{\text{Problem solving in c}\}}$ we will store string like this:-

P	r	o	b	l	e	m	\0
S	o	l	v	i	n	g	\0
u	s	i	n	g	\0		
c	\0						

Memory Representation.

PPS (02/Feb/2023)

- No of rows = no of strings, but the no of columns will equal the longest string length.
- The memory allocated to all the strings will be the size of the longest string causing memory wastage

② By using Pointers

We need to create an array of pointers pointing to strings, that's why we create char. This way all the strings are stored elsewhere in the exactly needed memory, And the pointers in the array points to those memory location causing no memory wastage

The pointers in the array points to the 1st character of the string.

Syntax:-

Data type* name[] = {"value 1", "value 2"};

Eg:- char* string[] = {"string 1", "string 2"};

Representation in Memory

H	I	\0	U	P	\0	A	I	\0
0			1			2		

for eg:- char name[5] = {"abcd"};

name → | a | b | c | d | \0 |

2040 200

3/Feb/2023

Read string from the user

Eg ① scanf() to read a string.

```
# include <stdio.h>
# include <conio.h>
int main()
{ char name[20];
  Printf("Enter name");
  Scanf("%s", name);
  Printf("your name is %s", name);
  getch();}
```

output

Enter name: Dennis
Your name is dennis

Eg (2)

```
# include <conio.h>
# include <stdio.h>
int main ()
{ char ch[11]: { 'p', 'r', 'o', 'b', 'l', 'e', 'm', '\0' },
char ch2[12] : "problem";
Printf ("char array value is %s", ch),
Printf ("string literal is %s", ch2),
getch (),  }.
```

o/p
problem
problem

Eg (3)

```
# include <conio.h>
# include <stdio.h>
int main ()
{ char name [30] = "Problem solving"),
  Printf ("%s", name),
  getch (),  }.
```

Eg (4)

```
# include <conio.h>
# include <stdio.h>
# include <string.h>
{ char name[30],  printf ("enter your name, "),
  gets(name),
  puts (name),
  getch (),
  }
```

output
Enter your name : JARVIS
JARVIS JARVIS JARVIS

Important String functions

There are many important string function when it's defined in string.h library.

S. Number	function	Description
01	strlen ()	Return the length of the string name.
02	strcpy ()	Copy the content of source string to destination string.
03	strcat ()	concat's or join first string the result of the string is store of in first string.
04	strcmp ()	Compares the 1st string with 2nd string if both string are same it returns 0.
05	strrev ()	returns reverse string.
06	strlwr ()	return string in lowercase.
07	strupr ()	returns string characters in upper case.

PPS (03/02/2023)

strlen() (i)

```
# include<stdio.h>
# include <conio.h>
# include <string.h>
int main()
{ char name[50], int length;
  Printf("Enter your name"); get(name);
  length = strlen(name);
  Printf("Your name has %d characters\n", length);
  getch();
  }.
```

o/p:- Enter your name Problem.
Your name has 7 characters.

strlen() (ii)

```
# include <stdio.h>
# include <conio.h>
int main()
{ char a[50], int i;
printf("\n Enter your name");
  get(a);
  i=0
while (a[i] != '\0')
  i++;
  printf("\n Total no of character = %d", i);
  getch(); }
```

	0	1	2	3	4	5	6
a	R	A	M	E	S	H	\0

strcpy()

```
# include <stdio.h>
# include <conio.h>
void main()
{ clrscr();
  char stud-name[30], emp-name[30];
  printf("Enter name");
  gets(stud-name);
strcpy(emp-name, stud-name);
  printf("student name = %s\n",
         Employment = %s\n", stud-name, emp-name);
getch(); }
```

o/p= Enter name= abcd , stud name = abcd
emp name : abcd ,

strcat()

```
#
#
#
void main()
{ char first-name[20], second-name[20];
printf ("Enter the 1st name:"),
gets (first name),
Printf ("Enter 2nd name:"),
gets (second name),
strcat (first-name, second name),
Printf ("Your full name = %s", firstname),
getch(), }.
```

o/p:- Enter the 1st name:- JARVIS
Enter second name :- KUMAR.
Your full name is JARVIS KUMAR,

strcmp()

```
#
#
#
void main()
{ char first-name[30], second-name[30],
int check;
Printf (" Enter first name:"),
gets ("first-name),
Printf ("Enter second-name"),
gets (second-name),
Check = strcmp (first-name, second-name),
if (check==0)
Printf ("Both string are equal"),
else if (check>0)
printf (" first string is greater than other"),
getch(),
}.
```

PPS 06/Feb/2023

(Searching & sorting)

- Searching is the process of finding some particular elements in the list.
- if the element is present in the list then the process is called successful, and the process returns the location of that elements, otherwise the search is called unsuccessful.
- There are 2 types of searching.
  (i) linear search (ii) Binary search

linear search

- Its also called as sequential search algorithm.
- In linear search we simply traverse the list completely and match each element of the list with the item whose location is to be found.
- if the match is found then the location of the item is returned, otherwise algorithm returns NULL in this.
- Here the list of array is traversed sequentially and every element is checked.

(Algorithm for linear Search)

step 1: Begin
step 2: set a [5]
step 3 : set $i \to 0$
step 4 : Input searching item
step 5: Repeat step 6 & 7 while $i<5$
step 6: if a[i] = item then print item found & location = i & exit.
step 7: $i \to i+1$
step 8: if $i \geq 5$ then print item not (404)
step-9: stop or End.

Ex:

	0	1	2	3	4
a	10	20	30	40	50

Program:-

```
#include stdio.h>
#include<conio.h>
 void main()
 {
int a[5]={10,20,30,40,50}.
  clrscr();
Printf("Input searching item");
scanf("%d",&item);
while(i<5)
{ if(a[i]==item)
  { printf("item found location=%d",i);
   break; }
   i++; }
  if (i>=5)
 printf("item not found");
   getch(); }
```

* Program take input from user & find location.

```
#include<stdio.h>
#include<conio.h>
 void main()
 {int a[5], i=0, j, item;
  clrscr();
 printf("Enter Array elements");
  for(j=0; j<5; j++)
  { scanf("%d",&a[j]);
    }.
Printf("Input Searching item:");
scanf("%d",&item);
 while (j<5)
 { if (a[j]==item)
  { printf("item found location=%d",i), break; }
   i++; printf("item not found"); getch(); }.
```

(PPS) (06 Feb 2023)

(Binary search)

- it is a searching algorithm for finding an elements position in a sorted array
- There are 2 methods to implement binary search algo's
  ① iterative method ② Recursive method.
- The recursive method of Binary search follows the ÷ & conquer approach.
- we have the array the data in particular order before searching operation. after that we find the middle element of array and compare with target element.
- we have to use the below formula to calculate the mid of the array $\boxed{mid = \frac{(begin + end)}{2}}$
- if item not found then again check the target element is greater than or less than middle element, if it's greater than we search right side of middle element else we search left side.

#(Algorithm for Binary Search)#

step1: Begin
step2: set a[5] // Array should be sorted (Ascend or Descend)
step3: set beg → 0, End → 4, f → 0
step4: Input searching item
step5: Repeat step 6 & 8 while(beg <= End)
step6: $\frac{beg + End}{2}$ // set mid.
step7: if a[mid] = item then set f = 1 & break;

Step 8: if a[mid] < item then
set beg → mid+1
else
set end → mid-1

Step 9: if f = 1 then
print item found with loc = mid
else
print "item not found"

Step 10: Exit

7/feb/2023. (Execution Process of Binary Search)

0	1	2	3	4	5	6	7	8	9
5	9	17	23	25	45	59	63	71	89.

∴ Searching item

Case 1: item = a[mid]
return mid.

case 2: item > a[mid]
mid+1

case 3: item < a[mid
mid-1

+ begin value change
− end value change.

begin	End	mid
0	9	$0+9/2 = 4.5$
5	9	$\frac{5+9}{2} = \frac{14}{2} = 7$
5	6	$\frac{5+6}{2} = 11/2 = 5.5$
6	6	$\frac{6+6}{2} = \frac{12}{2} = 6$

*Program

```
# include <stdio.h>
# include <conio.h>
void main ()
{ int a[5] = { 10, 20, 30, 40, 50 }, beg=0, end=4, f=0, mid, item;
clrscr();
printf ("Enter searching item");
scanf ("%d", &item);
while (beg <= End)
{ mid = (b+E/2)
if (a[mid] == item)
{ f=1; break; }.
```

```
if (a[mid] < item)
  beg = mid+1;
  else
  end = mid-1; }
  if (f==1)
printf ("item found with location = %d", mid);
  else
  printf ("item not found");
  getch();
  }.
```

#(Program to sort array elements in ascending and descending order)#

```
#
# void main()
  { int a[5], j, i, temp;
  printf ("enter array elements");
    for (i=0; i<5; i++)
     { scanf ("%d", &a[i]); //input (21345)
      }
    for (i=0; i<5; i++) //i=0
    { for (j=i+1; j<5; j++) // j=1
      { if (a[i] > a[j] //a[0] > a[1] [i<2>1]
        { temp = a[i];
          a[i] = a[j];
          a[j] = temp;
        } } }
        printf ("array elements:");
          for (i=0; i<5; i++)
        { printf ("%d", a[i]); } //12345
          getch(); }.
```

Here, this is using comparision of i values & j values with eachother.

PPS (08/02/2023)

Sorting

Arranging data in Ascending or descending is known as sorting

(i) **Bubble sort** (IMPORTANT)

- it arranges N number of array elements by placing the biggest element on proper position.
  * Working Ascending o/p descending
- it always arranges the Data in descending order.
- Bubble sort is one of the simplest algorithms that compares the adjacent elements and swap them if they are lesser/greater based on the conditions mentioned.
- Here each element is compared with all other elements in the array till its final place is found and No swap is required.
- This move of comparision of elements is known as pass

#(Working Process of Buble sort)#

- Compare the 1st and 2nd element of the array.
- if the 1st element is greater than second element then swap the elements.
- next compare the 2nd & 3rd elements. if the 2nd element is greater than the 3rd element swap them.
- repeat the steps untill the array is sorted.
- 2 elements are compared if they are in correct position then no swapping done otherwise will be done (swap).

#(Procedure / Execution Process of Bubble Sort)#

- result of this will always be in ascending order. but process will always be in descending order.

Pass① n=5 (iterations)

7 6 3 4 1
6 7 3 4 1
6 3 7 4 1
6 3 4 7 1
6 3 4 1 [7]

PPS (08/feb/2023)

6 3 4 1 } Pass ②
3 4 1 6

3 4 1
3 4 1 } Pass ③
3 1 4

3 1
1 3 } Pass ④

(Program)

```
#include<stdio.h>
#include<conio.h>
void main()
{int a[50], i, j, n, t;
printf("Enter elements 'no' in the list");
scanf("%d", &n);
printf("Enter the elements");
for (i=0; i<n; i++)
scanf("%d", &a[i]);
printf("Before Bubble sorting the elements are");
for (i=0; i<n; i++)
printf("%d/t", a[i]);
for (i=0, i<n-1; i++)   // Passes (outer loop)
{ for (j=i+1; j<n; j++)   // for comparison (inner loop).
if(a[i]>a[j]
{ t=a[i];
a[i]=a[j];
a[j]=t; } }}
printf("after Bubble sort elements are:");
for (i=0; i<n; i++)
printf("%d/t", a[i]);
getch();
}
```

PPS (9th Feb 2023)

Insertion Sort

It is a sorting algorithm that places an unsorted element at its suitable place in each iteration.

(Working Process of insertion sort)

- loop over positions in the array starting with index 1.
- The first element in the array assumed to be sorted.
- Take the second element & store it seperately in variable temp
- Compare temp with the 1st element if the 1st element > than temp variable then temp is placed in front of the 1st element
- Now the 1st 2 elements are sorted take the 3rd elements & compare it with the elements on the left of it, if there is no element smaller than it, then place it at the beginning of the array.
- Similarly place every unsorted elements at its correct position.

Execution Process

0	1	2	3	4	5
8	4	1	5	9	2

Sorted ← → unsorted.

temp: 4

0	1	2	3	4	5
4	8	1	5	9	2

← unsorted.

temp: 1

0	1	2	3	4	5
1	4	8	5	9	2

temp: 5

0	1	2	3	4	5
1	4	5	8	9	2

temp: 9

0	1	2	3	4	5
1	4	5	8	9	2

temp: 2

0	1	2	3	4	5
1	4	5	8	9	2

0	1	2	3	4	5
1	2	4	5	8	9

← Sorted →

We assume index 0 as sorted by Default.

Program/Shippet

```
n=6
for(i=1; i<n; i++)
{ temp = a[i];
  j = i-1
while (i>=0 && a[j] > temp)
{ a[j+1] = a[j]
  j--
  a[j+1] = temp; //for store
}
```

PPS (9th feb 2023)

(Full Program of insertion sort)

```
# include <stdio.h>
# include <conio.h>
void main ()
{
int a[100], i, j, n, temp;
printf ("enter no of elements");
scanf ("%d", &n);
printf (" enter integers");
for (i=0; i<n; i++)
{ scanf ("%d", &a[i]);
}
for (i=1; i<n; i++)
{ temp = a[i];
  j = i-1;
while (j>=0 && a[j] > temp)
{ a[j+1] = a[j];
  j = j--;
}
a[j+1] = a[j];
}
for (i=0; i<n; i++) // for print
{ printf("%d", a[i]);
}
getch();
}.
```

Programming for problem solving

Q What do you mean by sorting? WAP in C to sort n the integers using Bubble sort also draw the flow chart for the same?

Q Explain selection sort technique for sorting problem also write an Algorithm for selection sort, sort the following numbers using selection sort technique; [26, 54, 64, 19, 77, 31, 44, 55, 20]

Difference between Linear & Binary search

Linear search	Binary search
1.) In Linear search array (data) may or may not be sorted. i.e. it is not necessary that data should be sorted.	1.) In Binary search array (Data) should be sorted.
2) Item to be searched is compare with every element of array starting from first location.	2) Item to be search is compared with middle value of array.
3) Time complexity of linear search is $O(n)$	3) Time complexity of binary search is $O(\log_2 n)$
4) Its less efficient than binary search	4) Its more efficient than linear search.

Storage class.

function

Call by value

Call by refrence

Actual/formal parameter.

function call, define —

1-D Array → Sum

Counts.

2-D Array → multiplication

Sorting - Binary Search

Bubble.

Recursion - factorial.

Selection sort :- In selection sort the smallest value among the unsorted element of the array is selected in every pass & inserted to it appropriate position into the array.

The first element is selected from the unsorted array and placed at the first position after that second element is selected and placed in the second position the process continue until the array is entirely sorted.

Working process of Selection sort

- set min to the first location
- Search that minimum element in the array.
- Swap the first location with the minimum value in the array.
- Assign the 2nd element as min.
- Repeat the process until we get a sorted array.

Execution process of selection sort

7	4	10	8	3	1

```
# include -
# include -
int main()
{ int a[10] =
  int n=10; int i, j, min & swap;
for (i=0; i<n-1; i++)
{ min = i;
  for (j = i+1; j<n; j++)
{ if (a[j] < a[min]
  { min = j; }
  }
  if (min != i)
  swap (a(i), a(min)) }
  for(i=0; i<n; i++)
printf ("%d\t", a[i]);
getch();
}
```

initialy we consider of that the sorted published array empty no element in there.

0	1	2	3	4	5
7	4	10	8	3	1

Pass1

1	4	10	8	3	7

Pass2

1	3	10	8	4	7

Pass3

1	3	4	8	10	7

Pass4

1	3	4	7	10	7

Pass5

1	3	4	7	8	10

(Structure & union) # PPS (13/feb/2023)

- Structure is a user defined data type that combines/stores logically related data items of different data types together
- The 'struct' keyword is used to define a structure data type in a program.

Syntax

```
struct [structure name]
{ type member1;
  type member 2;
  ------
  ------
  type member N;
};
```

[Declaring a structure variable]

- You can declared structure variable in two ways by struct keyword within main function.
- By declaring variable at the time of defining structure

(FT 2) MOST INP (Array of Structure through keyboard.)

```
#include<stdio.h>
#include<conio.h>
  struct Employee
  { int id_no;
    char name[20];
    char designation[20];
    int salary;
    };

main()
{ employee Emp1;
  printf("Enter the id-no:");
  scanf("%d", &emp1.id.no);
  printf("Enter name:");
  scanf("%s", Emp1.name)
```

Declare of Globally

Variable name.

```
printf("\n Details of Employee");
printf("\n Employee id : %d", Emp1.idno);
printf("\n Employee name: %s", Emp1.name);
Printf("\n Employee salary %f", Emp1.salary);
Printf("\n Employee Designation: %s", Emp1.Desg);
getch();
}
```

Most IMP # (create student Records) # IMPORTANT (CT2)

```
# include <stdio.h>
# include <conio.h>
struct student
{ int id;
char name[20];
int marks;
};
main()
{ struct student s[5];
int i;
for (i=0; i<5; i++)
{ printf("\n id_no: ");
scanf("%d", &s[i].id);
printf("\n name: ");
scanf("%s", &s(i).name);
printf("\n marks: ");
scanf("%d", &s[i].marks);
}
printf("\n Data of all students");
for (i=0; i<5; i++)
{ printf("\n id.no: %d \t name: %s \t
marks: %d", s(i).id, s[i].name
s[i].marks); }
getch();
}
```

PPS (13-feb-2023)

(Union)

Union also stores the different types of elements. 'union' keyword is used to define union.

(syntax)

```
union    union-name
{

  };
```

14/feb/2023

Eg:-
```
#include<stdio.h>
#include<conio.h>
    struct student
    {int id;
     int roll_no;
     float percentage
      };
     void main()
     { struct student s1;
       printf("size of x = %d", sizeof(st));
       getch();
       }
```

[Enumerated Data type]

(DT)

- it is a user defined data type that consists of integer values and it provides meaningful names to these values.
- The use of enumerated (DT) in c makes the program easily to understand maintain.
- 'ENUM' keyword is used to define enumered (DT)

(Keywords are in small always)

:(Syntax):-

Enum Enumerator-name{ integer const1 ---- integer const N};

Nothing in b/w except a space gap

Example:-

enum fruits{Mango, apple, grapes};

• The default value of Mango is '0', Apple is '1', grapes is '2' if we want to change these default values, then we can do as given below.

(CT:- short notes + Example)

```
Enum fruits
{ mango = 2;
  apple = 1;
  grapes = 3; };
```

:[Enumerated Type declaration]:-

- As we know that in C language we need to declare the variable of a predefined types such as int, float, char etc.
- similarly we can declare a variable of a user defined data type such as (Enum).
- suppose we create the enum of type week as shown below.

→ Enum week {Sunday, monday, Tuesday, wednesday, --- };

- Now we create the variable of week type.

→ Enum week w;

- To create a variable the above two statements can be written as:

→ Enum{sunday, monday. ------ } w;

Eg:- Enum weekdays{sun=1, mo, Tue, wed, Thu, fri, sat};

```
int main()
{
Enum weekdays w; // declare variable of weekdays type.
  w = monday; // assigning value of monday to wedn.
printf("the value of w is %d", w);
getch();
}.
```

PPS (14/feb/2023)

Eg-2 Enum months { Jan=1, feb mar, ... Dec };

```
int main()
{
for(int i = Jan; i<=december; i++) // print the values of months
{
printf("%d", i);
}
getch();
}
```

output:- 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12

Most IMPORTANT

[Passing Array to the function]

- As we know that the array name contains the address of the 1st element.
- Here we must notice that we need to pass only the name of the array in the function which is intended to accept an array.

Syntax:- function name (array name); //passing array.

```
# include <stdio.h>
# include <conio.h>
void fun (int a[]);
void main()
{ int i, arr[5] = {1,2,3,4,5};
fun (arr);
printf("Now array elements are:");
for (i=0; i<5; i++)
{ printf("%d \t", arr[i]); }
getch(); }
void fun( int a[])
{ a[0]=6; }
```

Name of Array.

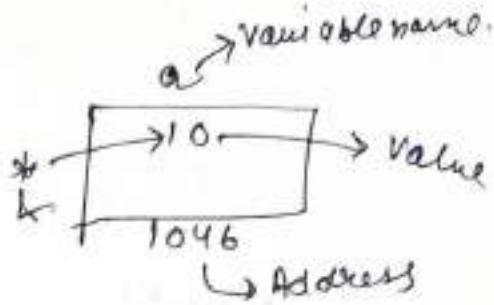
Pass Array.

(PPS) 14/feb/2023

(Module-5)

(Pointers)

- The pointer is a variable which stores the address of another variable.
- The variable can be of type int, char, array, function or any other pointer.



Syntax

data type *variable name;

Eg:

```
int a=10;
int *ptr;   // pointer to integer.
```

↑ Name of pointer.

```
ptr = &a;
printf("%d", ptr);   //prints Address
printf("%d", *ptr);  //prints value.
```

- By the help of '*' (dereference operator) we can print the value of pointer variable (Ptr).

[Declaring a Pointer]

- The pointer can be declared using '*' symbol.

Eg: [int *a], [char *c]

(Applications of Pointer)

- Dynamic Memory Allocation: in C language we can dynamically allocate memory using Malloc(), Calloc(), realloc(), free(). where the pointer is used.
- Arrays, functions, structures: pointers are widely used in arrays, functions & structures it reduces the code and improves the performance

(PPS) (14 Feb 2023)

:(Introduction to Dynamic Memory Allocation):-

- The concept (DMA) in C language enables the C programmer to allocate memory at runtime.
- Dynamic memory allocation in C language is possible by 4 functions (Stdlib.h) header file.
  - malloc()
  - calloc()
  - realloc()
  - free()
- Before learning above functions lets understand the diff b/w static & Dynamic memory allocation

(SMA)	(DMA)
• Memory is allocated at compile time	• Memory is allocated at run time.
• memory cannot be increased while executing program.	• memory can be increased while in the executing of program.

15/feb/2023

#(Methods used for Dynamic Memory Allocation)#

1) Malloc ():	stands for memory allocation. it allocates single block of requested memory.
2) Calloc ():	stands for contiguous allocation. Allocates the multiple blocks of requested memory.
3) Realloc():	stands for reallocation, it reallocates the memory occupied by Malloc(), calloc()
4) free():	it frees the dynamic allocated memory

PPS (15/feb/2023)

2) # (Malloc () function) #

- Malloc () is a built-in function declares in the header file <stdlib.h>.
- Malloc () method in C is used to dynamically allocate a single large block of memory with the specified size.
- it returns null memory is not sufficient.

(Syntax)

```
ptr = (caste-type *) malloc (byte-size)
```

eg. • ptr = (int *) malloc (n * size of (int))

Program:-

```
# include<stdio.h>
# include<conio.h>
# include<stdlib.h>
int main()
{ int i, n, *ptr;
printf("Enter total no of Values");
scanf("%d", &n);
ptr = (int *) malloc (n* size of (int));
printf("Enter Values");
for (i=0; i<n; i++)
{ scanf("%d", (ptr+i));
}
printf("The entered values are");
for (i=0; i<n; i++)
{ printf("%d\t" * (ptr+i)); }
free (ptr); //memory now released.
getch();
}.
```

Value of i gets added to ptr

PPS (15/feb/2023)

2) # (calloc())

- The calloc() allocates multiple blocks of requested memory.
- it initially initialise all byte to 0.
- it returns null if memory is not sufficient.

(Syntax)

ptr = (caste · type*) calloc (number , byte · size)

eg :- Ptr = (int *) calloc (n, size of (int)),

Program :-

```
# include <stdio.h>
# include <conio.h>
# include <stdlib.h>
int main ()
{ int i, n, *ptr,
printf ("Enter total no of values"),
scanf ("%d", &n),
Ptr = (int *) calloc ( n, size of (int)),
printf ("Enter values"),
for (i=0; i<n; i++)
{ scanf ("%d", (ptr+i)),
}
printf ("The entered values are"),
for (i=0; i<n; i++)
{ printf ("%d \t" *(ptr+i)), }
free (ptr),
getch(),
}
```

[Very Void Pointer]

- malloc doesn't have an idea of what it is pointing to.
- it Allocates memory requested by the user without knowing the type of data to be stored inside the memory.
- The void pointer can be typecasted to an appropriate type

Realloc ()

Syntax

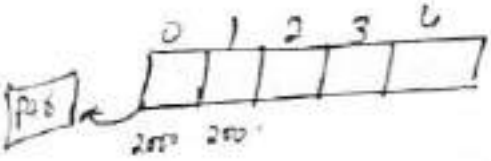
```
void * realloc (void * ptr , size-t new size)
```

ptr → Pointer to the previously allocated memory

new size → new size

Eg

```
#
#
int main ()
{
int n, i, * ptr ;
printf ("Enter total no. of values : ");
scanf ( "%d", &n);
ptr = (int*) malloc (n* sizeof(int));
printf("\n Enter the values : ");
for (i=0; i<n; i++)
  {
  scanf("%d", (ptr+i));
  }
printf("\n Enter updated size of n : ");
scanf ("%d", &n);
int *ptr1 = (int*) realloc ( ptr, n* sizeof(int));
printf("\n previous address : %d, new address : %d", ptr, ptr1);
printf("\n The Entered values are : ");
for(i=0; i<n; i++)
{printf ("%d\t", *(ptr + i));
}
```



→ Previously allocated mem is both in this ptr

```
getch ();
}
```

E.g

free()

```
#
int *display ()
{
int n, i, *ptr
ptr = (int *)malloc(3* sizeof(int));
printf("\n Enter the values:");
for (i=0; i<3; i++)
{
scanf("%d",(ptr+i));
}
return ptr;
}
int main ()
{
int i, *ptr1;
ptr1 = display ();
printf("\n The Entered values are ");
for(i=0; i<3; i++)
{
printf("%d\t", *(ptr1+i));
}
free(ptr1);
```

3) Realloc()

(PPS) 15/Feb/2022

- If memory is not sufficient for Malloc() & calloc() you can reallocate the memory by realloc() function.
- In short it changes the memory size

(Syntax)

int * ptr1 = (int *) realloc (ptr, n * sizeof(int));

4) free()

- The memory occupied by Malloc() & calloc() function must be released by calling free function otherwise it will consume memory untill program executes as exited.

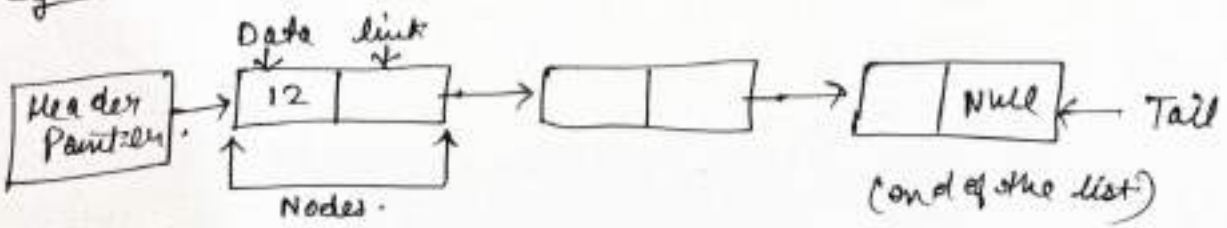
(Syntax)

free (ptr);

(Linked list)

- A linked list is a linear Data structure.
- Here elements are not stored at a contiguous location.
- The elements are linked using pointer.
- A node contains two fields that is a Data Section, Address section. that holds the address of the next element in the list.
- The size of the linked list is not fixed. & data items can be added at any location in the list.
- The last node of the list contains pointer to the NULL.

Eg:-



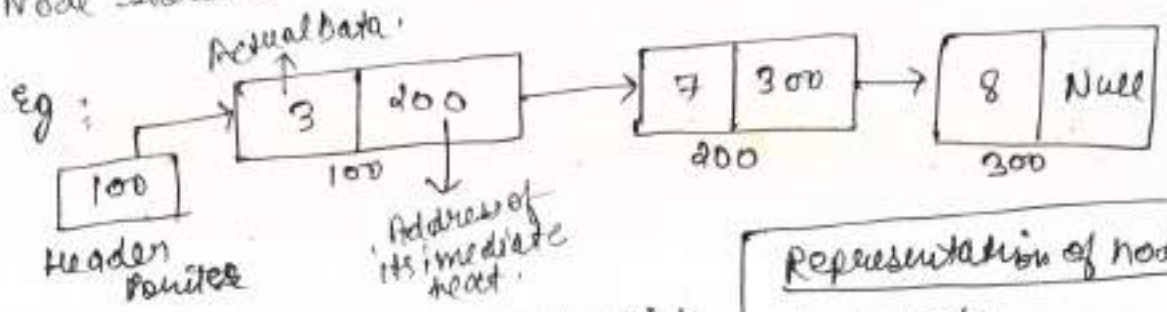
(PPS) 15/Feb/2023

Types of linked list

① Singly LL: Here we can move or traverse the (LL) in only one direction.

- A node here consists of 2 parts (Data Part & link Part)
- Data part of the node stores actually information that is to be represented by the node while the link part of the Node stores the address of its immediate successor

Eg:



* this arrow represents the link

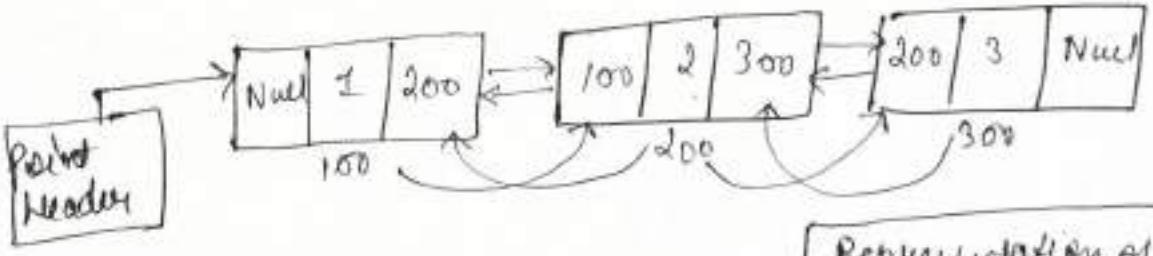
Representation of node in Prog

```
struct node
{ int data;
  struct node *next;
};
```

THIS WILL BE USED IN SELF REFERTIAL

② Doubly LL: In this type of linked list we can move or traverse the (LL) in both directions (forward & backward direction.

each node has 3 parts. (1-Data Part, 2-Address Parts).

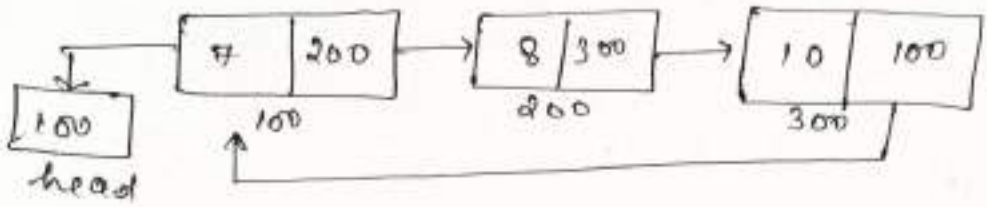


Representation of node in Prog

```
struct node
{ int data;
  struct node *next;
  struct node *prev;
};
```

③ **Circular LL**

- In this type of LL the last node of the LL contains the list of the first head node of the link list in its next pointer.



Most IMP

Use of pointer in self referential structure

- A structure can have members which points to a structure variable of the same type.
- These types of structures are called self referential structures and are widely used in dynamic data structure like linked list trees etc.

Syntax

```
struct structure_name
{ int data;
  struct node *link
};
```

- Here link is a pointer to a struct node variable. Structure a referential (self) DS is essentially a structure defination which includes at least one member that is a pointer to the structure of its own kind.

Eg:

```
struct node
{ int data;
  struct node *next;
};
```

Points to the same type of structure.

PPs (16/Feb/2023)

eg ①:

```
struct code
{
int i;
char c;
struct code *ptr;
};
int main()
{ struct code vari1;
  struct code vari2;
  vari1.i = 65;
  vari.c = 'A';
  vari.ptr = 'Null';
vari2.i = 66;
vari2.c = 'B';
vari2.ptr = Null;
vari1.ptr = &vari2;/
printf("%d %c", vari1.ptr->ptr1, vari1.ptr->c);
```

* operator use of to called the variables.

PPS (18/Feb/2023)

(file handling)

- file is a collection of logically related data that a PC treats as single unit.

- PC stores file to secondary storage so that the content of file remains when the computer is turned off.

- when a computer reads a file it loads file from secondary storage to main memory. when it writes to a file it transfers data from main memory to secondary storage

- file handling in C, enables us to create, update, read & delete the file stored on local file system

(functions for file handling)

There are many functions in C library to open, read, write, search & close the file. A list of file functions are given below :-

Function	Description
1) fopen()	:- opens new or existing file
2) fclose()	:- closes the file
3) fprintf()	:- writes data into the file
4) fscanf()	:- reads data from the file
5) fputc()	:- writes a character into the file
6) fgetc()	:- reads a character from file.
7) fseek()	:- sets the file pointer to given position.
8) fputw()	:- writes an integer to file
9) fgetw()	:- reads an integer from file
10) ftell()	:- used for reading the current position of pointer.
11) frewind()	:- sets the file pointer to the begin of it,

:(Opening Modes of C in Standard I/O Program):-

Mode in Program	Description
r :-	open a text file for reading content.
rb :-	open a file for reading the content in binary mode.
w :-	open a file for writing the content
wb :-	open a file for writing content in binary.
a :-	open a file in append mode
a+ :-	open a text file in read & write mode.
ab :-	open a binary file in append mode
r+ :-	open a text file in read & write mode
wb+ :-	open a file for both read & write in binary.
ab+ :-	open a binary file in read & write mode
rb+ :-	open a binary file in read & write mode

:(Types of Files):-

There are 2 types of file :-

(i) Text file (ii) Binary.

(Text file:)

- These are the most basic & simplest type of file that a user can create in a C program.
- We can create a text file using an extension (.txt) with the help of a simple text editor.
- In general we can use note pad for the creation of .txt files
- These files stores information internally in ASCII format but when we open these files the content/text opens in a human readable format.
- Text file consumes a very large space in storage.

(Binary files)

- These files stores information & data in binary format (0's & 1's) (The binary number system). Thus the file occupy comparatively lesser space in storage.
- in simpler words the binary file stores info & data the same way a PC holds the info in its memory.
- The binary files are created with the extension dot bin (.bin) in a program.

:- [Built-in function for file handling] :-

There are many built in functions that are used to perform file handling operations.

1) fopen() = This function is used to open a file to perform operations such as reading, writing etc. In a C program we declare a file pointer & use the function fopen() to open a file for performing various operations

Syntax:-

```
FILE * fp.
fp= fopen("filename", "mode");

perform operation.
fclose(fp);
```

2) fclose() :- This function is used to close a file

(Program) (fputc())

```
#
#
main()
{ FILE * fp; //Creating a file pointer.
fp= fopen("file.txt", "w"); // opening file.
```

```
fputc ("a", fp);   //writing single character
fclose (fp);  // closing file
getch();
}.
```

Program② $\boxed{\text{fprintf()}}$

```
//
# main()
  FILE * fp;
  char str [50];
  int a = 10;
  char ch = 's'
fp = fopen ("abc.txt", "w");
  if (fp == NULL)
  { printf ("error occured");
    exit (1);
  }
printf ("enter the string:");
  gets(str);
fprintf (fp, "%d %s %c", a,
        str, ch);
fclose (fp);
  getch();
}.
```

FPUTC() = writing

→ file

→ single character.

→ for the function.

$\boxed{\text{fp = file pointer.}}$

Program③ $\boxed{\text{fputs()}}$

```
#
# main()
{ FILE * fp;
fp = fopen ("myfile.txt", "w");
  if (fp == NULL)
  { printf ("Error");
    exit(1);
  }
fputs ("Hello C programming", fp);
fclose (fp);   getch();  }
```

PPS (17/feb/2023).

fgetc()

```
#
#
main()
FILE *fp;
char c;
clrscr();
fp=fopen("student.txt","r");
while (c=fgetc(fp)!=EOF)
{ printf("%c",c);
}
fclose(fp);
getch();}
```

fscanf()

```
#
#
int main()
{ FILE *fp;
char s[25];
fp=fopen("file.txt","r");
while(fscanf(fp,"%s",s)!=EOF)
{ printf("%s",s);
}
fclose(fp);
getch(); }.
```

(fgets())

```
#
#
void main()
 FILE *fp;
char text[300];
clrscr();
   = fopen("myfile.txt","r");
  if (fp == NULL)
  {printf(" error occured");
    return0;
    }
while (fgets(text, 300, fp)) != EOF
{ printf("%s", text);
  fclose(fp);
  getch();
    }.
```

Q.WA C Program

Command Line Arguments in C

* The arguments passed from command line are called Command Line Arguments.
* The Arguments are handled by main() function.
* To use the command line Arguments, we must define main() with two arguments as follows:

```
int main (int argc, char * argv[])
```

* argc → Count the number of arguments passed.
* argv[] → is a pointer array, points to Each argument passed to the program.
  * The argc counts the file name as the first argument
  * In argv the first argument is the file name always. The argv contains the total number of arguments.

* It is mostly used when you need to control your program from the console.

* argv[0] holds the name of the program.
* argv[1] points the first command line argument.

E.g

```
# include <stdio.h>
# include <conio.h>
Void main (int argc, char * argv[]) // argc & argv is variable name
{
 int i;
 clrscr();
 printf ("\n No. of Arguments %d", argc);
 for (i=0; i< argc ; i++)
 {
  printf ("\n argument : %s", argv[i]);
 }
 getch();
}
```

Array of pointers

O/P

no. of argument 6
argument : C:\ Path (TURBOC3 \ SOURCE \ COMM.EXE
argument : a
argument : b
argument : c
argument : d
argument : e

File Name & consider as a argument

NOTE : In Turbo C go RUN → Arguments
Then Write Argument seperated by space.

NOTE → You can Pass arguments to the main function only when you are calling your program using Command line.

* If you pass many arguments within double quote, all arguments will be treated as a single argument only.

Eg

If you pass as a argument

"Hello c Programming"

The o/p is -

no. of argument 2

argument : C : \TurboC3\SOURCE\COMM.EXE

argument : Hello c Programming.

→ File name

→ Consider as 1 argument

Standard C pre-processors

* C Pre-Processor Directives

* As the name suggests, Preprocessors are program that process our source code before compilation.
* Preprocessor directives are executed before compilation.
* All preprocessor directives starts with hash # symbol.
* There are four main types of Preprocessor Directives:

1) Macros
2) File Inclusion
3) conditional compilation
4) other directives.

① Macros

* A Macro is a piece of code in a program that is replaced by the value of the macro.
* Macro is defined by # define directive
* The syntax of the Macro is given below

```
# define PI 3.14
```

Preprocessor Directive → # define
Macro name → PI
Macro value → 3.14

* There are two types of Macros

① object-like Macros
② Function like Macros

① object-like Macros

* The object-like macros is an identifier that is replaced by value.
* It is widely used to represent numeric constants.

E.g

```
# include <stdio.h>
# include <conio.h>
# define PI 3.14
main ()
{
printf ("%f", PI);

getch ();
}
```

O/P

3.14

② Function like Macros

The function like Macros looks like function call.

E.g

```
# include <stdio.h>
# include <conio.h>
# define MIN(a,b) ((a) < (b) ? (a) : (b))
void main ()
{
printf ("Minimum between 10 and 20 is : %d\n",
        MIN(10,20));
getch ();
}
```

O/P

Minimum between 10 and 20 : 10

② File Inclusion Directives

* This type of preprocessor directives tells the compiler to include a file in the source code program.

* This is a process of inserting External files containing functions or macro definitions into the C program.

This avoids rewriting those functions or macro definitions

* These files can be included as:

include "file name" E.g # include "Grade.c"

File Name

* This directives searched for the header file in the current directory.

* The directory of current input files is called the current directory.

③ Conditional Inclusion Directives or Conditional Compilation

* There are several Directives that allow you to selectively Compile portions of your program's source code.

* Conditional selection is rarely performed using #defined values.

* Conditional Compilation directives are a type of directive that helps to compile a specific portion of the program or to skip the compilation of some specific part of the program based on some conditions.

* This can be done with the help of the preprocessing commands —

#if, #Endif, #Else, #Elif

* <u># if</u> Evaluates a constant integer expression or condition.

* If the constant expression #if is true, the code that is between it and #Endif it is completed. otherwise, the intervening code is skipped.

* The #Endif directive marks the end of an #if block.

<u>E.g</u> A simple example of #if and #Endif directive.

```
# include <stdio.h>
# include <conio.h>
# define M 100
int main ()
{
 # if (M > 99)
 printf ("\n compiled for array greater than 99");
 # Endif
 getch ();
}
```

* If the condition satisfies, the code will execute.

② #else

* The #else directive works much like the else that is part of the c language.
* It establishes an alternative if #if fails.

Eg A simple Example of #else directives.

```
#
#
#define M 100
int main()
{
#if (M>99)
printf("\n compiled for array greater than 100");
#else
printf("\n compiled for small array");
#endif
getch();
3
```

③ #elif

* The #elif directive means "else if" and establishes and if-else chain for multiple compilation option.
* If the expression is true, that block of code is compiled and no othe #elif expressions are tested.
* otherwise, the next block in the series is checked.

```
#if Expression
 Statement sequence
#Elif Expression 1
 Statement sequence
#Elif Expression 2
 Statement sequence
- - - - - -

- - - - - -

#Elif Expression n
 Statement sequence
#Endif.
```

(4) Other Directives

* Apart from the above directives, there are two more directives that are not commonly used. These are :

(1) **#undef Directive** ⇒ It is used to undefine an existing Macro.

E.g. #undef MACRO

* This directive is used to remove the definitions related to a particular macro.

(2) #pragma Directives

* This directive is a special purpose directive and is used to turn on or off some features.
* This type of directives are compiler specific i.e. they vary from compiler to compiler.

* #ifdef and #ifndef

* Another method of conditional compilation uses the directives #ifdef and #ifndef, which means "if defined" and "if not defined", respectively.

* #ifdef macro-name
  statement sequence
  #Endif

* If Macro-name has been previously defined in a #define statement, the block of code will be compiled.

* #ifndef macro-name
  Statement sequence
  #Endif