20/9/2022

* Embedded system :- It is the combination of both Hardware & software.

⇒ Embedding a software inside a Hardware to perform a single task or multiple task based on requirement.

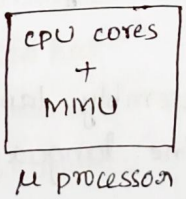
⇒ The Hardware boards are based on Micro controller, Micro processor and system ON chip (SOC).

⇒ Micro controllers based devices are designed to perform single task at a time (dedicated task).

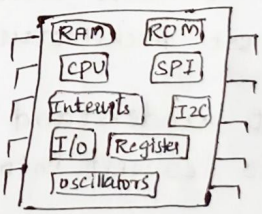
- They operate at Mega Hertz (MHz)
- MC's will have limited RAM, ROM & IO resources.

⇒ Micro processors operates on Giga Hertz (GHz)

- They are designed to perform multi tasking.
- This can be achieved by multiple CPU ~~resource~~ cores & operating systems (os). (Quad core)
- In Micro processor's it will have CPU cores along with memory management unit (MMU).



μ processor



μ controller.

past → 16 bit
32 bit (X-86 Architecture)

present → 64 bit
Future → 128 bit

CISC Based Architecture.

⇒ For processor based devices we have memory in terms of Giga bytes (GB).

⇒ All other peripherals including hard disk, RAM are placed on a board (Mother board).

⇒ System on chip (SoC) based on ARM Architecture of 32/64 bit.

- It consists of every connectors like SPI, I2C, Interrupts, UART, I/O etc.. except RAM & ROM.

⇒ ARM company will develop only CPU core logics
- → Advanced RISC (Reduced instruction set computer) Machine.
- → Application oriented (General purpose)
- → Real time application oriented
- → Micro controller oriented (single task)

⇒ CPU can understand only machine language (or) Machine instructions (or) Binary language (0,1).

- whether it may be a controller, processor or SoC. It will works on NPN logics. These NPN transistors are constructed by billions of logic gates. As they requires 0's & 1's as the input.

⇒ Users cannot understand this machine language, so we can't write the code in machine. so, we need another language. so, architecture vendors developed assembly language ISA (Instruction set Architecture).

⇒ The CPU cannot understand this assembly language. we requires assemblers to convert into machine language.

⇒ 'c' is the architecture independent language (or) portable language.
- 'c' is structured user friendly programming language.
- CPU cannot understand c pgmg language. we require compilers to convert c language to machine language.

⇒ compilers are developed by Architecture vendors (or) silicon (develops main architecture) vendors (or) product vendors
- Develops chipset
  Eg:- Qualcom.
- ↳ Develops o/p req product
  Eg:- One plus
  Real me
  Redmi Etc

⇒ Machine instructions consists of opcode & operand.

'c' (code) → compiler → Binary language

clanguage → changes to → Assembly language → Assembler (changes AL to ML) → Machine language (Binary)

⇒ Linux is open source which is derived from Unix operating sys.
- For linux we have to follow GPL (General public licence).
- According to this, anybody can download, modify, Rebuilt & Redistribute for free of cost.
- Linux was developed by linus torvalds.
- Linux kernal is written purely in 'c' pgmg language with some assembly codes.
- This linux will supports 20+ architectures because of its open source development.
- With a single command we can load & unload the device drivers without system reboot. It is more secure.
- Variants of linux operating systems.
  1. Ubuntu
  2. Red hat
  3. Fedora
  4. x-ubuntu
  5. Kali linux
  6. Open source
  7. Raspbian
  8. Android

  Etc

  Different vendors

  common linex kernal but GUI is different (appearance).

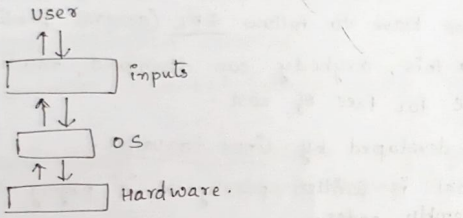
21/09/22

⇒ Operating system (OS) is an executable binary file which acts as a mediator between user & Hardware.

⇒ A user can provide the inputs in 2 ways. They are :-

① GUI (Graphical user Interface).

② CLI (command line Interface).



⇒ The GUI should have applications to take the inputs

⇒ The CLI should have specific commands.

⇒ Source code contains instructions in user understanding format that can be any programming language instructions i.e.,

- C instructions
- CPP "
- Py "
- Java "
- Python "

⇒ Applications, executable files (OS) contains machine instructions (Binary language) which are not in user understanding format.

⇒ CPU will always executes the instructions given by the user.

⇒ RAM (Random Access memory) is a temporary storage memory. We can access the data very fast. It is known as primary memory. It is volatile (As soon as power lost, the data will be lost).

⇒ Hard disk, SD card, pendrives (ROM) are permanent storage devices. It is known as secondary memory. These are Non-volatile in nature (The data will be present even though the power is lost).

⇒ This memory is used to store user info, applications, executable files, commands etc.

⇒ If CPU has to execute any application it must be loaded from Hard disk to user space of RAM.

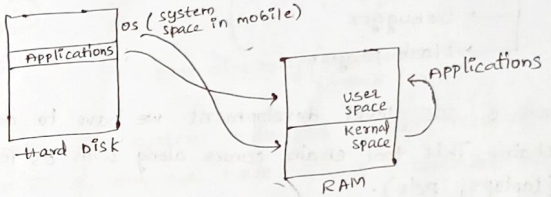
⇒ RAM is divided into two types.

① User space of RAM.

② Kernal space of RAM.

⇒ The applications are loaded into user space of RAM.

⇒ The OS is loaded into Kernal space of RAM.



⇒ CLI is faster in execution (w.r. doesn't include GUI).

⇒ The commands are developed in 'c' language only & they are compiled to executable binary.

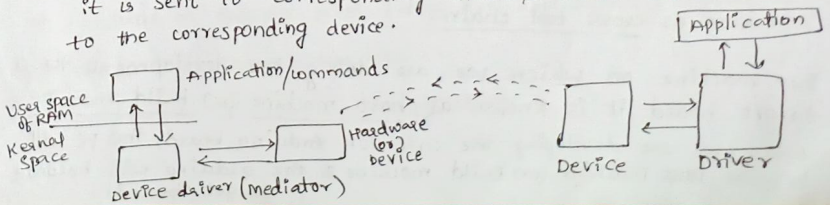
⇒ This commands doesn't have any instructions related to GUI.

⇒ Kernal is the main core component of linux operating system which maintains subsystems like :-

① process Management
② Memory "
③ File "
④ I/O "
⑤ Network "

} along with + Device drivers. to service the request coming from user interface & Hardware interface

⇒ The user input taken from the application, from there it is sent to corresponding driver & from driver it is sent to the corresponding device.



⇒ system bus consists of (Data + Address + control).

⇒ For proper communication we should have application, device driver & device. (H/W)

⇒ For Micro controller based devices we have to download Ide tools (Integrated Development Environmental tools).

IDE
- Editor
- Compiler
- Hex converter
- Simulator
- Debugger
- Flash program

} From user manual.

(gcc) (VI)

⇒ For processor & SOC level development we have to download the tool chain. This tool chain comes along with OS for host machines (laptops, PC's).

⇒ In this tool chain we have editor, compiler, debugger, loader & other programming language supporting tools.

⇒ For controllers, processors, SOC we have common set of protocols (rules) and other controllers [UART, SPI, I2C].

Serial communication protocols
- UART
- I2C
- SPI
- USB
- CAN
- ⋮

Parallel I/O ports
- → Leds
- → sensors
- → Keyboards
- → Buzzers
- → Displays etc

controllers
- → UART master
- → I2C controller
- → SPI controller etc.

⇒ The tool chain that is downloaded for the target boards, it is known as cross tool chain.

⇒ The machine on which we are doing the development for a target board it is known as host machine (or) build machine.

eg:- When we are developing the code for Arduino board, the PC will become Host machine (or) Build machine & the arduino will become target board.

22/9/22

NTFS → New technology file system
FAT32 → File Allocation Table
ext4 → extended file system.

⇒ Every OS will have a file system, windows will support NTFS, FAT32. Linux will support FAT32, FAT60, VFAT, ext4, ext3, ext2.

⇒ The default file system is ext4 for linux system.

⇒ The Hireachy of the Folders arranged in a particular format order to store/get the information.

⇒ For every user OS will allocate separate work space for the user with user name by home directory.

/home/Engineer
↳ root directory (super user work space).

⇒ The ext4 file system mounted under root directory ('/').

⇒ under file system we have /boot ⇒ contains kernal image
- Vmlinuz, Vmlinux
- system.map
- configuration file
- initrd

⇒ /dev (device files/device nodes).

⇒ /bin, /sbin, /usr/bin ⇒ commands/tools.

⇒ /proc ⇒ process info

⇒ /media ⇒ pendrive/USB drive

⇒ /mnt ⇒ To mount or connect

⇒ /usr ⇒ For current user info

⇒ /etc ⇒ contains username's, passwoads & other info

⇒ /lib ⇒ It consists of libraries & kernal modules

⇒ /include ⇒ Header files (user level applications & system related)

⇒ /usr/src ⇒ Kernal headers.

⇒ /usr/include ⇒ User level applications.

$ sudo apt update
$ sudo apt upgrade

↓ Requesting super user permission.

⇒ From terminal application these permissions are executed.

⇒ To install the applications in Ubuntu, we have Ubuntu software centre.

⇒ The command used to install is

sudo apt install < ____ >
↓
Package Name
(or)
Tool name.

commands / tools / packages

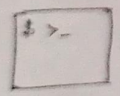
GCC ⇒ compiler
vim ⇒ editor

⇒ Package name & application name are different from each other.

⇒ when you open the terminal it opens the current user work space.

⇒ engineer @ engineer. Thinkpad - - - - : ~ $
↓ current user
→ Tilde symbol
→ shell prompt
↓ Home directory of current user

Terminal app



→ command / binary

⇒ Default locations for commands

(bash) /bin
/sbin
/usr/bin
/usr/local/bin

⇒ Terminal application internally runs Bash executable file. Under this terminal we can execute the commands which are present in /bin
/sbin
/usr/bin
/usr/local/bin.

⇒ Bash shell was developed by Brian Fox for the GNU project as a free software replacement for the Bourne shell first released in 1989.

⇒ "pwd" displays the present (or) current working directory path.

⇒ "cd" command changes the directory.

⇒ "cd /" jump to root directory.

⇒ From any directory we can jump to 2 default directories i.e., Home directory & root directory.

cd / → / → root directory

cd ~ → ~ → Home directory

⇒ "." represents current working directory.

⇒ ".." represents previous or parent directory.

⇒ Files names, Folder names are case sensitive.

⇒ while creating a directory (or) File, don't use spaces in between file names (or) directory names.

⇒ ls command will list out files & folders present in current working directory.

⇒ ls -l command will list out type of files with permissions.

Type of File	permissions: owner	group	other	No. of links	owner name	Group Name	size	Time stamps	File name
d	r = read = 4								
l	w = write = 2								
c	x = execute = 1								
b									
p									

d = directory
l = symbolic link
c = character device file
b = Block device file.
p = pipe

To change permissions,
$ chmod 777 <file name>.

⇒ This information we can see in GUI from properties of Files.

⇒ File command will provide the info about the type of file.

⇒ For a Normal textual file we don't have executable permissions.

⇒ grub = second stage of boot loader
(or)
Grand unified boot loader.

⇒ when we power off the system the file system contains only the default information.

⇒ /proc, /sys, /dev Folder content will get vanished after power off the system. It will get back after powering on the System.

23/09/2022

For manual pages

$ command --help

$ man (2) <system call>

$ man < >
↓
command name
library fu^n name
tool name

⇒ Tool chain even supports for other programming languages like Java, python, perl, c++ & other debugging languages (including c)

⇒ The Tool chain is installed under /usr/bin. It is native tool chain.

⇒ Editors are used to write the programs even to see the content & modify also.

⇒ The size of the file depends upon number of characters written into the File. Each character will take 1 byte memory.

100 characters = 100 bytes

⇒ Every 'c' pgm is saved with ".c" extension.

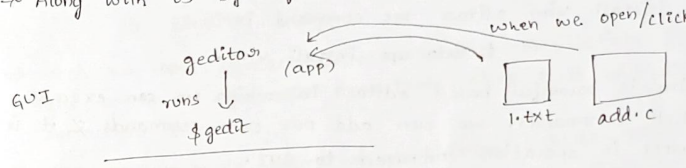
⇒ compiler will start the compilation process, if it is .C file only.

⇒ CPU cannot understand 'c' statements. It can only understand machine instructions (Binary language). So, we need compilers to convert 'c' statements to machine instructions.

⇒ compiler, assembler, assembly language are architecture dependent. But 'c' is architecture independent programming language.

⇒ we have two types of editors. They are CLI based & GUI based.

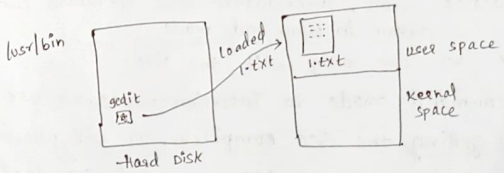
⇒ Along with os by default we get geditor application.



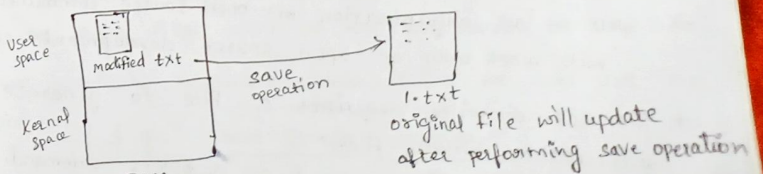
⇒ gedit :- we can open the textual file from GUI or CLI

⇒ From GUI interface we have to click on the textual file.

⇒ From CLI we have to use the command gedit < File name >.
↓↓
gedit 1.txt.



⇒ The File is opened under user space of RAM through geditor application & whatever the modification we do, it will not reflected to the original file until we perform save operation



⇒ The CLI based editors are opened under command line interface.

④ Nano editor (default editor for linux os with limited operations)
- It is a small, basic CLI based editor which will have less options.

④ Vi editor (visual interface)
- Improved version of vi editor is Vim editor.

⇒ To install vim editor the command is
$ sudo apt install vim.

⇒ Vim is powerful based editor in which we can execute shell commands, we can add our own commands & it is faster in operation compared to GUI.

⇒ Vim editor has 2 operating modes
① command mode
② Insertion mode

④ Even Nano editor operates in 2 modes

⇒ By default it will open in command mode. In this we can perform operations like save, quit, search, replace, undo, redo, copy, paste, cut, delete, scroll, Intendation. These operations can be done in command mode.

⇒ In insertion mode we can only edit the file.

⇒ To change from command mode to insertion mode, we use i

⇒ In linux we are getting the GCC compiler. It will present in /usr/bin. It is also known as GCC command, Gcc tool. It is an executable file.

⇒ GCC is GNU 'c' compiler.

⇒ GNU is an organisation (or) open source community which will work only on open source development. eg:- linux unix

⇒ The GCC compiler requires 'c' file to generate single executable binary file.
- To compile the file use $ gcc sample.c command.

⇒ The executable file is generated with a default name is a.out.

⇒ We cannot have two files with same name in same folder.

⇒ The process of converting 'c' statements to machine instructions is known as compilation. The compilation has 4 stages :-
① Pre processing stage.
② compiling stage.
③ Assembling stage.
④ Linking stage.

⇒ when we have syntactical errors, the compilation process fails, we don't generate executable files & it will displays syntactical error.

⇒ when we write 'c' pgm we must follow syntax. when we have syntactical file we dont get executable files it is compilation error.

⇒ during compilation process even we may get linking errors.

⇒ syntax's are verified in 2nd stage of compilation.

Compile process

$ gcc sample.c
↓
① pre processing stage → sample.i (c-code)
② compiling stage → sample.s (assembly code) x-86
③ Assembling stage → sample.o (object code) Machine instruction
④ Linking stage
↓
a.out (Machine instructions).

⇒ sample.i, sample.s, sample.o are known as intermediate files we don't see these files when we run command (gcc sample.c)

⇒ when we want to generate these files we can use the command :-

$ gcc -save-temps sample.c.

24/9/22

Windows ⟶ Task manager.

Linux (Ubuntu) ⟶ system monitor
↓
contains info in /proc.

$ PS - (snap shot)

⇒ The process which are there in b/w square braces are kernal threads [Kworker/1:2].

⇒ Current working process info can be seen by ps-ef command.

⇒ Error Message format :-

- It shows path / c prgmg File name / line / func^n name / errors msg.

* Bugs
  - → compilation error
  - → False output.
  - → Run time error.

⇒ How to generate executable file with desired name.

Ⓐ gcc -o executable File name c prgm File name

(or)

Ⓐ gcc c prgm File name -o executable File name.

⇒ Execution can be done by terminal only. To run the program we need to specify the path along with executable file name.

Ⓐ path / executable File.

⇒ $ ./a.out. → Executable File name.
↓
current working directory

⇒ This internally invokes the loader to load the a.out file into user space of RAM for execution, now the control is given to the CPU. CPU will start executing the main function.

⇒ For every user level 'c' pgm main() function is the entry point and the main() function termination is the exit point.

⇒ As soon as the main function completes the execution, the executable file is automatically removed from user space of RAM.

⇒ The errors which are generated during the program execution time by terminating the program they are ntg but "Run time errors"

⇒ To debug these Run time errors we have to use gdb tool i.e., GNU debugger. This can be used only on user level pgms.

⇒ This can be applicable only on user level executable files which contains debug information.

⇒ Compilation errors are different from Run time errors.

⇒ compilation errors occurs while converting 'c' statements into machine instructions if there are any syntax errors.

⇒ Even gdb can be used to debug false outputs.

⇒ Every operating system will have an executable file format. For linux it is ELF (executable linking format).

⇒ In Vim editor after writing program, use

:w = write / save

:x (or) :wq = save & quit

⇒ To jump to a particular line use the command,

:(line number) ⇒ :5 ⇒ Jumps to 5th line

(or)

line number gg ⇒ 3gg ⇒ Jumps to 3rd line

⇒ GG ⇒ Jumps to Top / bottom of the program.

⇒ yy ⇒ copy ; nyy ⇒ copy 'n' lines from cursor point.

⇒ P = paste, dd = delete, ndd ⇒ cut or delete 'n' lines.

⇒ u ⇒ undo, ctal+r ⇒ Redo

⇒ save every 'c' pgm with a meaningful name.

⇒ To search for a string / string name [/hi]

n → To search from top to bottom.

N ⇒ To search from bottom to top.

⇒ To search & replace

: %s/hi/HI → New string.

↓ old string

⇒

ctrl + F	= scrolls page by page (Forward)
ctrl + B	= scrolls page by page (Backward)
ctrl + U	= scrolls in upward direction for half page
ctrl + D	= scrolls in downward direction for half page

⇒ = no. of lines ⇒ allign with indentation.

⇒ when we write 'c' pgms we should follow some standards.

(*) Structure of C program :-

- Indentation should be followed.
- write 'c' pgm description on the top as a comment (License info ⇒ GPS).

// ⇒ single comment

/* ... */ ⇒ Multiple comments.

① program description :-

- License info (GPL)
- Developer's info (mail id)
- structures description / functions description / union [defination]

comments — For user understanding purpose.

⇒ The description will tells its importance
- Important logics description.

⇒ comments are used for user (or) developer understanding purpose but not for the system (or) CPU.

⇒ comments are not part of the executable file. they are removed in the first stage of compilation.

② Header Files :-

include < ... .h > → /usr/include.

↓ pre-processor directive.

⇒ compiler will search for the header file in default directory (/usr/include).

⇒ " ... " → It will search for header file in current directory.

⇒ Even we can include sample.c file code from

include "sample.c".

③ Macro definations :-

def MAX constant/code

↑ (macro name)

⇒ In General, in header files we will write

⇒ used for verification purpose during compilation process

⇒ not a part of pgm

[so, all these will include in header files]

- → Function declarations
- → structure / union definations.
- → Macro definations
- → conditional compilation statements
- → type def

⇒ These are used only for verification purpose during compilation process. Once verification is done these statements are removed. These verifications are done in second stage of compilation.

④ structures/union definations

⑤ Global variables declarations & definations.

⑥ Function declarations.

⑦ Function definations.

⇒ In libraries (ready made) will have function definations in pre-compiled or executable format.

⇒ If we dont link (or) include required libraries (or) header files compiler will generate linking errors.

⇒ For user level 'c' pgm main() function is the entry point.

⇒ In a 'c' pgm we can't define 2 functions with same name.

⇒ In a complete 'c' pgm we can have only one main() fun^{c} definition.

⇒ The program logic must be constructed within the main() function only. The program logic can be divided into multiple functions & these functions are invoked (or) called from main function or its dependency functions

⇒ The CPU will start executing the main function. As soon as the main function is completed, the program will be terminated automatically.

26/9/22

✱ ASCII character set :- (American standard code for information interchange).
(0-127)

⇒ we have total 128 characters

⇒ Each character will take 1 byte memory.

⇒ Each character will have ASCII value. This ASCII value is stored in binary format in that 1 byte memory.

⇒ With the help of these ASCII character set. we are going to write the 'c' pgms. This ASCII character set we can see on our keyboard.

⇒ Each ASCII character is enclosed within the pair of single quotes [' '].

⇒ The range of ASCII value lies between (0-127) only.

⇒ The characters are

- Upper case → 'A' ------ 'Z'
  ASCII Value ⇒ 65 90
- Lower case → 'a' ------ 'z'
  97 122
- Symbols (operators) → # $ & ; ' " ! ? < > + - = * ----
- Executable characters → '\n', '\v', '\t', '\r' -----
- Numeric characters → '0' to '9'
  48 57

⇒ Each symbol will have their own importance in 'c' pgmng language.

⇒ some symbols are used as operators → [& | ^ ---- AND OR ex-OR]

& some are used as syntactical representations [; : , ( ) { } --- delimeters]

⇒ '0' ≠ 0
48 / 0
Character constant zero (1 byte) · Integer constant zero (4 bytes)

✱ Executable characters :-

⇒ Non printable characters
⇒ changes cursor position
⇒ ASCII values
⇒ standard 'c' library output functions.

'\n' → new line character
'\v' → Vertical tab
'\t' → Horizontal tab
'\r' → Cariage return
'\a' → Alert
'\b' → Back space.
'\f' → Form Feed
'\0' → Null character

⇒ Null character :- It's Ascii value is zero

⇒ It is used at the end of string constants.

⇒ It acts like termination of string constant.

'a'	"a" (a\0)	0xa
Character constant (1 byte)	string constant (2 bytes)	integer constant (4 bytes)

↳ when we use (" ") string constants null character automatically includes.

⇒ `$ man ascii` ⇒ It will displays the ascii character set with decimal, Hexa decimal & octal format.

Delimeters :-

⇒ Delimeters are used in 'c' prgmg for syntactical representation

⇒ If we don't use these delimeters at appropriate location in 'c' pgm it will generate syntactical (or) compilation error.

⇒ { } = pair of curly braces are used to write a ~~blog~~ block statements & also for initilization.

⇒ ( ) = paranthesis is used to write expressions & functions (while passing & receiving the inputs).

⇒ [ ] = square braces are used at array declaration & definations

⇒ : = colon is used for defining a lable (switch case).

⇒ ; = Semi colon is used at end of every 'c' stmt.

⇒ , = comma acts as a separator.

⇒ Delimeters are verified in 2nd stage of compilation.

Keywords :-

⇒ Keywords are predefined words whose meaning is already known to the compiler. These are standardized according to c88, c89 standards.

⇒ Our gcc compiler will supports c88, c89 standards.

⇒ If we understand these 32 keywords then we can understand 'c' language.

c standards

gcc compiler follows c88, c89 standards
- c88 → 32 keywords
- c89 → 37 "
- c11 →

char, int, float, double	Basic data types	volatile	if	+ bool, inline, restrict, Imaginary, complex
long, short	Modifiers	const	else	
signed, unsigned	sign qualifiers	typedef	while	
auto, register, extern, static	storage classes	sizeof	do	
		void	for	
		struct	break	
		enum	continue	
		union	goto	
			return	
			switch	
			case	
			default	→ control statements

⇒ we should not use these keywords as Identifiers

Identifiers :-

⇒ user defined words in a 'c' pgms are identifiers.

⇒ c tokens include keywords, constants, Identifiers, strings, special symbols & operators etc.

⇒ Identifiers should be a meaning ful name
→ variable name, Array name, function name, file name, macro name, label name, structure name etc.

⇒ Rules for the identifiers :-

⇒ It is the combination of upper case, lowercase & Numeric constants.

⇒ It should not start with numeric.

⇒ It should start with underscore (_) or letter.

⇒ we should not use keywords as identifiers

⇒ Identifiers are case sensitive.

⇒ All keywords are written in lower case

⇒ we should not use spaces in b/w the identifiers.

⇒ Identifiers should not exceed more than 32 characters.

⇒ The latest compilers they are recognizing even more than 32 characters.

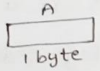
⇒ # constants :-

⇒ It will represents the type of data.

⇒ We have character constants ①, string constants ②, integer constants ③, real constants ④.

*① **character constants** :- A letter or a symbol enclosed with in the pair of single quotes (' ') that represents a character set. Entire ASCII character set comes under character constant.

⇒ By default compiler will decide 1 byte memory for a character constant. In this 1 byte memory it will store corresponding ASCII character's ASCII values in binary format.



ASCII code 'A' = 65

2 | 65
2 | 32 - 1
2 | 16 - 0
2 | 8 - 0
2 | 4 - 0
2 | 2 - 0
1 - 0

1000001

⇒ 1 byte = 8 bits ✓

⇒ 1 nibble = 4 bits ✓

② %c ⇒ converts binary to decimal & decimal to equivalent ASCII character

'A'

| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |

⇒ 1 byte

⇒ printf ("%c", 'A');

⇒ %c format specifier always reads the data from 1 byte memory.

⇒ To store a character constant we require 1 byte memory we can request the memory by using character variable.

⇒ data type will provide

→ size (1 byte) 8 bits = (0-7)

→ Range of values [based on sign qualifiers]

signed char = -128 to +127 $(-2^7 \text{ to } 2^7-1)$

Unsigned char = 0 to 255 $(0 \text{ to } 2^8-1)$

⇒ syntax for declaring a variable :-

< Qualifiers < modifiers < sign qualifiers < data type < variable name

Key words. Identifier

⇒ Variable is nothing but it is the name given to that memory location.

⇒ Based on initilized values, variables are of 2 types.

① Initialized variables ✓

② Uninitialized variables ✓

① when the memory is allocated & initilized with some values it is known as initialized variables. (int x = 5 ;)

② ⇒ when only the memory is allocated but not initialized with any value is known as uninitialized variables. (int x ;)

⇒ compiler will decide the memory for variable based on data type. The memory gets allocated when CPU executes the statement given by user (when we run the prgm) in User space of RAM (memory gets allocated)

· char x = 5 ; ⇒ 'x' is initialized variable cuz char = 1 byte memory is allocated & is initialized with '5' value.
(1 byte memory)

· int y; ⇒ 'y' is uninitialized variable, even though int memory of 4 bytes is allocated based on int data type but not initialized with any value.
4 bytes memory

✱ Declaration of character variable [uninitialized variable]

```
char x;
```

✱ Defining a character variable [Initialized variable]

```
char x = 'A';
```

✱ Sign Qualifiers

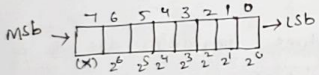
signed char = (-128 to 127) (-128 to 0, 0 to 127) = $(-2^7$ to $2^7-1)$

unsigned char = 0 to 255 = $(0$ to $2^8-1)$.

⇒ It will tells us the range of values.

⇒ Signed means both positive & negative values (or) numbers.

⇒ For signed qualifiers msb bit doesn't have place value. It is used to represent positive (or) negative value.



0 → positive
1 → Negative.

1 bit
Min = 0
Max = 1

7							
Msb	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0
0	1	1	1	1	1	1	1

→ 0 (min value)
→ 127 (max value)

Msb	6	5	4	3	2	1	0
1	0	0	0	0	0	0	0
1	1	1	1	1	1	1	1

→ Min value (-128)
→ Max value (-1)

⇒ If we don't apply any sign qualifiers, by default it will consider it as signed type.

⇒ For unsigned type there is no signed bit.

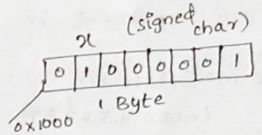
Msb
1 0 0 0 0 0 0 0 → min value/bit
1's ⇒ 0 1 1 1 1 1 1 1
2's ⇒ +1
1 0 0 0 0 0 0 0 → (-128) = Min -ve value

Msb
1 1 1 1 1 1 1 1 → Max bit
1's ⇒ 0 0 0 0 0 0 0 0
2's ⇒ +1
0 0 0 0 0 0 0 1 ⇒ (-1)
Max -ve Value

⇒ main()
{ char x = 127;
printf ("%d", x);
}
⇒ O/P
127

⇒ char x = 128 ⇒ O/P = -128
⇒ char x = 255 ⇒ O/P = -1
⇒ unsigned char x = 255 ⇒ O/P = 255.
⇒ %d (Decimal integer constant, reads only 4 bytes).
⇒ Irrespective of signed qualifiers, we can store the ASCII character in a character variable.
⇒ In a signed character variable we can store -128 to +127 integer values.
⇒ In an unsigned character variable we can store 0 to 255 integer values.

⇒ main()
{ char x = 'A';
printf ("%d - %c", x, x);
}



%c prints 'A'
%d converts binary to decimal = 65.

⇒ char x = 65;
printf ("%d, %c", x, x);
prints 65 (01000001)
prints A.

⇒ main()
{ char x = 129;
if (x > 127)
printf ("T");
else
printf ("F");

O/P
False
cuz x = 129
Given character is signed
x = 129 = -129
After conversion cuz signed range is from (-128 to 127)
∴ -129 > 127
False

→ we can use size of operator to know how many bytes are allocated. It is a built-in operator in compiler, it is written no. of bytes to be allocated.

⇒ gcc -v (To know compiler version) ⇒ 9.4.0

⇒ mode()
{
char x = 'A';
printf("%d %c", x, x); ⇒ 65 A
printf("%d", sizeof(x)); ⇒ 1 (1 byte).
}

27/09/2022

⊛ Integer constants :- (4 bytes)

⇒ we can represent in 3 systems

① Decimal (Base-10) (0,1,2,3,4,5,6,7,8,9) — 127

② Hexa decimal (Base-16) (0-9, A to F) — 0X127 — 2^4 → 4 binary bits

③ octal (Base-8) (0-7, 10 to ..) — 0127 — 2^3 → 3 binary bits.

⇒ By default compiler will decide 4 bytes for an integer constant.

⇒ The integer constant's equivalent binary gets stored in that 4 bytes memory.

⇒ In decimal system we can have both positive & Negative numbers. But in Hexa decimal & octal system we have only positive numbers.

⇒ %d format specifier is used to print both positive & Negative number system both in decimal system.

⇒ %d format specifier will always reads 4 bytes value.

• Decimal 127 | 0000 0000 | 0000 0000 | 0000 0000 | 0111 1111 |

• To known the size of bites ⇒ printf("%d", sizeof(127)); ⇒ 4

⇒ %u is used to print the output only in unsigned format.

⇒ If the integer variable is signed type we must use %d format specifier.

⇒ If the integer variable is unsigned type we have to use %u format specifier.

⇒ Data type

⊛ int
→ size = 4 bytes (32 bits) (0-31)
→ signed int = -2^{31} to $2^{31}-1$
→ unsigned int = 0 to $2^{32}-1$
→ Type of data = Integer constants (Decimal / Hexa decimal / octal).

② Hexa Decimal (Base-16) ↓ 2^4 each Hexa digit → 4 binary bits

⇒ This Hexa decimal is mainly used for debugging purpose cuz it can be easily converts Hexa to Decimal

Decimal	Binary	Hexa Decimal	Binary	octal	Binary
0	0	0X0	0000	00	000
1	1	0X1	0001	01	001
2	10	0X2	0010	02	010
3	11	0X3	0011	03	011
4	100	0X4	0100	04	100
5	101	0X5	0101	05	101
6	110	0X6	0110	06	110
7	111	0X7	0111	07	111
8	1000	0X8	1000	10	001 000
9	1001	0X9	1001	11	001 001
10	1010	0XA	1010	12	001 010
11	1011	0XB	1011	13	001 011
12	1100	0XC	1100	14	001 100
13	1101	0XD	1101	15	001 101
14	1110	0XE	1110	16	001 110
15	1111	0XF	1111	17	001 111
16	10000	0X1F	0001 1111	20	010 000
17	10001	0X2F	0010 1111		

0x127

0000 0000	0000 0000	0000 0001	0010 0111

⇒ unsigned int x = -1;
printf ("%u %d %x", x, x, x);

(A) %u = $2^{32}-1$
%d = -1
%x = FFFFFFFF
%x } converts decimal to Hexa

0000 0000 0000 0000 0000 0000 0000 0001
2's ⇒ (-) 1
0000 0000 0000 0000 0000 0000 0000 0000
1's ⇒
1111 1111 1111 1111 1111 1111 1111 1111
F F F F F F F F

⇒ Unsigned int x = -1;
int x = -1;
printf ("%d %u", x, x);
if x > 0
printf ('T');
Else
printf ('F');
%d = -1
%u = $2^{32}-1$

⇒ printf ("%x", 0x127); ⇒ 127
printf ("0x%x", 0x127); ⇒ 0x127
printf ("%d", 0x127); ⇒ 295
printf ("%d", 127); ⇒ 127
printf ("%x", 127); ⇒ converts 127 into Hexa decimal = 7F (decimal)
printf ("%o", 0x127); ⇒ 447

0000 0000	0000 0000	0000 0001	0010 0111
		4	4 7

⇒ $(0x127)_{16} = (295)_{10} = (447)_8$

(*) Octal (Base-8)
2^3 ⇒ Each octal digit = 3 binary bits

0127

0000 0000	0000 0000	0000 0000	0101 0111

⇒ Printf ("%o", 0127); ⇒ 127
⇒ printf ("%d", 0127); ⇒ 87
⇒ printf ("%x", 0127); ⇒ 57

* $(127)_8 = (87)_{10} = (57)_{16}$

%o } unsigned
%o }

2^6 2^5 2^4 2^3 2^2 2^1 2^0
0 1 0 1 0 1 1 1
64+16+4+2+1 = 87
0101 0111
5 7 = 57

⇒ main()
{
unsigned x = 65;
printf ("%c %d \n", x, x);
printf ("%d", size of (65));
}
↳ 4 bytes

ⓐ compilation error
ⓑ runtime error
ⓒ A 65
ⓓ None of the above

⇒ If we don't specify the data type, but we specify sign qualifiers by default compiler will consider it as int data type.
⇒ If we use more pointers, Run time error will occur.
⇒ If we don't specify sign qualifiers then it is compilation error.

main()
{
unsigned char x = 0x41424344;
printf ("%c \n", x);
}

⇒ It will take only right side 2 bits (LSb)
44 ⇒ equivalent binary
0100 0100
$2^6+2^2 = 68$
"%c" prints its equivalent ASCII character
⇒ D

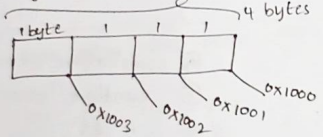
⇒ main()
{ unsigned int x = 0x41424344;
printf ("%c\n", x);
}

O/P ⇒ 68 ASCII character ⇒ "D"

⇒ Memory is a collection of bytes. Each byte will have address. Users can access the memory location in two ways:-

① By using name given to the location

② By using the address.



⇒ The Byte order storage also depends on architecture. we have 2 formats:-

① Little Endian
② Big Endian

⇒ X-86 will support Little Endian format

⇒ ARM will support both little Endian & Big Endian. But it supports and enable little Endian defaultly.

(*) The Least significant byte is assigned to least significant address, it is known as Little Endian format.

(*) The least significant byte is assigned to most significant address it is known as Big Endian format.

⇒ As X-86 follows little Endian format.

41	42	43	44

Big Endian

44	43	42	41

⇒ Big endian byte order storage is used in n/w communication

Little Endian

unsigned char x = 0x41424344;
printf ("%c", x);

It considers 44

0100 0100
2^6 2^2

$2^6+2^2 = 68$

⇒ It prints 68 equivalent ASCII character = (D)

Big Endian

unsigned char x = 0x41424344;
printf ("%c", x);

It considers 41

0100 0001
2^6 2^0

$2^6+2^0 = 65$

⇒ It prints 65 equivalent ASCII character = (A)

⇒ main()
{
int x = 259; (4 bytes)
char y; (1 byte)
printf ("%d\n", y);
}

O/P:- It prints 3

2|259
2|129 – 1
2|64 – 1
2|32 – 0
2|16 – 0
2|8 – 0
2|4 – 0
2|2 – 0
1 – 0

100000011

(3)

⇒ main()
{
int x = 'A';
printf ("%d\n", x); ⇒ 65
Printf ("%x\n", x); ⇒ 41
}

2|65
2|32 – 1
2|16 – 0
2|8 – 0
2|4 – 0
2|2 – 0
1 – 0

01000001
4 1

28/09/2022

⇒ Difference between program & process :-

- Program is nothing but it is an executable file.
- The program which is under execution in user space of RAM is known as process.

✱ Modifiers :-

⇒ we have modifiers to increase or decrease the size of a variable at the declaration or defining time.

→ short
→ Long.

→ short is applicable only on int data type which will reduce the size by 2 bytes.

✱ short int

⇒ size = 2 bytes
⇒ signed short int [Range = -2^{15} to $2^{15}-1$]
unsigned short int [0 to $2^{16}-1$]
⇒ Integer constant.

✱ Long int

⇒ size = 8 bytes
⇒ signed long int [-2^{63} to $2^{63}-1$]
⇒ unsigned long int = 0 to $2^{64}-1$

→ Long modifier :-

- It will be differ in 32 bit & 64 bit machine.
- In 32 bit machine it is considered as 4 bytes.
- In 64 bit machine it is considered as 8 bytes.
- This long modifier is applicable on int & double.

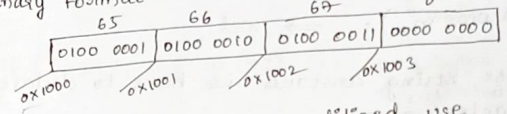
Long int → 8 bytes
Long double → 12 bytes = 32 bit Machine
16 bytes = 64 bit "

⇒ String constant :-

⇒ A group of characters enclosed with in a pair of double quotes "abc" is string constant.

"ABC" ⊛ {'A', 'B', 'C', '\0'}

⇒ Each character will take only one byte memory. In this 1 byte memory it will store corresponding ASCII value in binary format.



⇒ To know the size of bytes utilized use

```
printf("%d\n", sizeof("ABC"));
printf("%s", "ABC");
```

⇒ To display the binary data in string format we have to use %s format specifier in printf function.

⇒ %s format specifier always requires string's starting base address.

⇒ whenever we pass a string constant in a function, indirectly we are passing that string's base address.

```
printf("ABC")
```

- It will pass the info of string's starting base address (0x1000)

⇒ The strings which are passed to a function their memory will be decided by the compiler in read only data section (ro data) in your executable file.

⇒ The string size will depends upon the length of the string including null character (\0).

⇒ The length of the string is not fixed. It varies based on the user input. so, %s format specifiers in printf function it will start printing the data from that address until it finds null character.

⇒ %i can be integer constant / Decimal / Hexa decimal / octal.

⇒ %ld ⇒ long int

⇒ %lu ⇒ unsigned long int
unsigned long double.

⇒ printf function will start printing the data from the specified address until it finds null character.

⇒ printf("welcome\n"); ⇒ welcome

⇒ To store these string constants we have to declare or define character array.

⇉ Real constants :-

⇒ Real constants are ntg but floating point numbers. These floating point numbers can be stored in 2 types
- Single precision (float)
- Double precision (double)

⇒ By default real constants comes under double type (8 bytes)

⊛ 2.34

⇒ Float (single precision) (4 bytes) ↓ 6 digits after decimal point

2.340000

⇒ double (double precision) (8 bytes) ↓ [10-12 digits after decimal point]

2.340000000

2	.	3	4	0	0	0	0	0	0	0

⇒ In kernal programming we dont use any Real constants.

⇒ Float x = 1.23
double y = 1.23
x ≠ y

⇒ These real constants can be expressed in exponential format.

⊛ 0.02345
= 2.345000 * 10^{-12} ⇒ 2.345000e-2

⇒ printf("%e", 0.02345); ⇒ 2.345000e-2

⇒ printf("%e", 123.24); ⇒ 1.232400e+2

⇒ printf("%d\n", size of (1.23)); ⇒ 8 bytes (cuz real constants comes under double type).
(%ld)

⇒ printf("%ld", size of (1.23f)); ⇒ 4 bytes [cuz where we use f in single precision which are of 4 bytes (float type)].

⇒ Float x = 1.23
printf("%d", size of x); ⇒ 4 bytes (cuz variable x is of float type).

⇒ double y = 1.23
printf("%d", size of y); ⇒ 8 bytes (cuz variable y is of double type).

⇒ %f %F
%e %E
%g %G

⇒ %e format specifier will print the output in exponential format so the decimal part is adjusted to single digit & fractional part is printed upto 6 digits and the adjusted digits are printed in terms of powers of e.

⇒ %F ⇒ The decimal part is printed as it is & the fractional part is printed upto 6 digits
- printf("%f", 123.24); ⇒ 123.240000 (6 digits)

⇒ For %g, the overall o/p is printed upto 6 digits only.
- printf("%g", 123.24); ⇒ 123.240 (6 digits)

Variables :-

① Memory ?
② When the memory gets allocated
③ default values ?
④ Access ? (scope)
⑤ de allocation of memory.
⑥ Storage location ?

⇒ Variable :- It is the name given to that memory location.

⇒ Based on scope (nothing but accessibility) variables are of 2 types.

① Local variable (inside of the fun^c).
② Global variable (outside of the fun^c).

⇒ A variable which is declared or defined outside the fun^c is known as global variable.

⇒ A variable which is declared or defined inside the fun^c is known as local variable.

⇒ Based on initialization, these variables are classified as follows :-

- Uninitialized local variables
- Initialized local variables

scope :- Fun^c / block scope
default value :- Garbage value.

- Uninitialized global variables
- Initialized global variables

scope :- File scope
default values :- zero

* Uninitialized local variables :-

① Compilers based on data type will allocate memory.
② memory gets allocated during program execution time (when CPU executes that statement) in stack Frame of corresponding fun^c.

③ Uninitialized local variables contains garbage value. [This memory is earlier used by some other function, Now it is allocated for current fun^c].

④ The scope of these variables lies within the block or fun^c from where they are declared or defined.

⑤ These variables memory gets destroyed when the fun^c or block gets terminated.

⇒ The same set of rules are applicable for uninitialized local arrays,
- pointer variables,
- fun^c pointer,
- structure variable,
- void pointers
- union variables
- enums & ---

* For initialized local variables contains initialized values and same set of rules are applicable.

```
int y=5;  ⇒ initialized global variable
void fun()
{
  int x=9;  ⇒ initialized local variable
  ≡
}
```

```
int y;  → uninitialized global variable
void main()
{
  int x;  → uninitialized local variable.
  ≡
}
```

* Global Variables :-

⇒ Global variables memory is also decided by compiler based on data type & it decides the memory section during compilation time only.

⇒ The global variables memory gets allocated during program loading time.

⇒ Uninitialized global variables memory gets allocated in
- bss segment.

⇒ Initialized global variable's memory gets allocated in .data segment.

⇒ Global variables will have file scope from where they are declared or defined till the end of file.

⇒ These variables can be accessed in all the below fun^c From where they are declared or defined till end of the file.

⇒ These variables memory gets destroyed when process terminates.

⇒ we can't declare or define two variables with the same name with in same scope.

⇒ we can have 2 variables with the same name in different scopes.

* Types of scope

→ Block Scope → { }
→ Fun^c scope → main()
→ File scope → From starting to Ending
→ process " → Accessibility in total program.

⇒ By default all local variables comes under Auto storage class

NOTE :-

⇒ A global variable applied with constant qualifier, memory gets allocated in .rodata segment (compiler decides to go to ——) → where to go.

29/09/2022

⇒
```
main()
{
  char x = 0XFF;
  Printf ("%d", x);
}
```
o/p = -1

⇒
```
main()
{
  unsigned int x = 0XFFFFFFFF;
  printf ("%u", x);
  printf ("%d", x);
}
```
o/p
%u = $2^{32}-1$
%d = -1

⇒ unsigned char x = -1;
printf ("%d\n", x); ⇒ 255
printf ("%u\n", x); ⇒ 255

⇒ unsigned (or) unsigned int x = 0XFFFFFFFF;
printf ("%u\n", x); ⇒ $2^{32}-1$
printf ("%d\n", x); ⇒ -1
} (o/p will be same)

⇒ char x = 0X7F;
Printf ("%d\n", x); ⇒ 127
printf ("%u\n", x); ⇒ 127

⇒ A variable can be used in that particular fun^c after its declaration or definition.

⇒
```
main()
{
unsigned char x=0XFF;
printf("%d", x);
}
```
o/p = 255

⇒
```
main()
{
signed int x=0XFFFFFFFF;
printf("%u\n", x);
Printf("%d\n", x);
}
```
o/p :- %u = $2^{32}-1$
%d = -1

⇒
```
main()
{
  printf("%d\n", x);
  int x=10;
}
```

In this case we will get compilation error.
⇒ we need to declare the variable before printf function

⇒
```
void fun();
main()
{
  int x=10;
  printf("%d", x);
  fun();
}
void fun()
{
  printf("%d\n", x);
}
```

Life (or) scope

⇒ A variable which is declared in one func cannot be accessed directly in another func.

⇒ Memory allocation time to memory deallocation time of variable is known as life time.

⇒ scope is applicable only when there is life for variable.

⇒ scope of a variable is nothing but visibility of variable during its life time.

⇒ Even though we can have two variables with a same name in a 'c' pgm but their addresses are different.

⇒ Addresses are always printed in Hexa decimal format.

⇒ we can access a variables address as ampersent (&) followed by variable name ⇒ "& variable".

⇒ %p ⇒ for address.

⇒ Scope is not applicable for "addresses".

⇒ Scope is applicable for "variable names".

⇒ By using variable name (x) address we can access within that region also

④
```
main()
{ int x=10;
  {
  int x=20;
  printf("%d", x);
  }
}
```

scope & life of x=10

scope & life of x=20

scope & life of x=10

⇒ we can access the x=10 in x=20 block by using pointer but not using x as name.

⇒ The life of global variable is till end of the program but scope is till end of file.

①
```
main()
{
  int x=10;   (20)
  {
    x=20;
  int x=40;
  printf("%d\n", x);
    x=50;
  }
  printf("%d\n", x);
}
```

1st x = 10 → 20
2nd x = 20 (overwrites)
3rd x = 40 → 50 (overwrites 40 with 50 in this block only)

scope & life

→ prints 40

x=50's life is ended at }

→ It will print 20

∴ o/p = 40
20.

②
```
int x=10;
main()
{
  printf("%d\n", x);
}
```

o/p = 10

⇒ This global variable scope can be accessed below all the funcs from where they are declared or defined till the end of the file.

⇒ The life of global variable lies till program terminates.

⇒ scope of global variable is File scope

⇒ scope of local variable is func scope (or) Block scope.

⇒ The default value of uninitialized global variables is zero.

⇒ The default value of uninitialized local variable contains garbage value.

③

```
int x;
main()
{
  x = 30;
  int x = 40;
  printf("%d\n", x);
  {
    x = 50;
    printf("%d\n", x);
    int x = 60;
    printf("%d\n", x);
  }
  printf("%d\n", x);
}
```

x = 0 (0x1000) → 30 (0x1000) → overwrites

40 (0x2000) → prints (40)

50 → overwrites 40 (0x2000) → prints (50)

x = 60 (0x3000) → prints (60)

It prints (50)

scope of 0x1000

scope of 0x2000

scope of 0x3000

As same name is there, the scope will be according to address

⇒ In the scope of 0x2000, 0x1000 can't be accessed, only 0x2000 is accessed (visibility) as it is head for that func^{c}.

⇒ In 0x3000, 0x2000 can't be accessed so no visibility new one is formed for 0x3000 address.

③ ~~program~~ ⇒ If visibility is destroyed then it goes to the scope which is present. (As 0x3000 is present in 0x2000 can't be accessed). If 0x3000 is destroyed 0x2000 can be accessed. so, 50 is painted for last printf function as 0x3000 is destroyed & next visibility is 0x2000 & 0x1000 can't be accessed as next visibility is 0x2000

⇒ If we want more clarity about scope & life we can print the address by printf("%p\n", &x);

④

```
int x;
main()
{
  x = 30;
  int y = 40;
  printf("%d\n", y);
  {
    y = 50;
    printf("%d\n", y);
    int z = 60;
    printf("%d\n", z);
  }
  printf("%d\n", x);
}
```

x → stores zero for uninitialized global variables

x = 0 → overwrites 30

y = 40 → prints 40

y = 50 → prints 50

z = 60 → prints 60

z memory gets destroyed

It prints 30.

y, x memory gets destroyed

⑤

```
main()
{
  printf("%d\n", x);
}
int x = 10;
```

o/p

compilation error.

⑥

```
extern int x;
main()
{
  PF("%d\n", x);
}
int x = 10;
```

storage class

This command informs the compiler that variable x memory is defined somewhere, access it.

It prints 10

⇒ extern keyword is applicable only on global variables. For that strnt variable memory is not allocated.

* C Statements :-

Types of c statements

- simple c [variable declaration (or) definition]
- Expression
- control statements (selective or decision making loops or iterative stmts, jump).

* Standard c library i/p & o/p fun^c :-

⇒ Function :- Group of instructions defined to perform particular task.

⇒ Function name must be a meaningful names.

⇒ Fun^c name comes under identifiers.

⇒ A program logic can be divided in to multiple sub tasks. Each task defines a fun^c.

⇒ In general a fun^c will take the i/p's as arguments, so that fun^c will process the data based on i/p's & finally generates the required o/p.

→ For library functions manual pages

Syntax (or) command :- $ man 3 fun^c name

man 2 system call → for system calls

man command name → For tool or command.

⇒ Functions are of two types.

① User defined functions

② pre defined functions.

⇒ In general any function will have
- fun^c declaration
- fun^c definition
- Fun invocation / call.

⇒ For a user defined function we can see declaration, definition, invocation within the 'c' prgm.

⇒ For a pre defined functions declarations are present in header files, definitions are present in libraries, invocations from the program. ↳ (definitions will be in precompiled or executable format).

⇒ Libraries are of two types → static libraries, dynamic "

⇒ Dynamic libraries are also known as shared library (pre defined generally).

shared library ↳ libname.so (shared obj)

static library ↳ lib name.a — archieve.

Eg:- libc.so → c library

libm.so → math library.

⇒ compiler knows only standard c library functions.

⇒ when we are using other library functions in our prgm we must link that library during the compilation process, otherwise we get linking errors.

gcc test.c -l pthread

test.c : c prgm file name for compilation

-l pthread : ↳ library name

⇒ we must include header files in c prgm for compiler verification purpose.

⇒ How do we come to know about header files?

(*) we have to take the help of manual pages. In manual pages for that fun it shows header files to be included, fun^c declarations, no. of i/p's & their description, operation to be performed & return values.

⇒ Default path for header files is /usr/include.

30/9/2022

⇒ Every library fun^c internally invokes one system call.

⇒ Each system call will have an equivalent kernal call. Each system call is identified by a unique positive number. overall we have 300+ system calls.

⇒ printf fun^c is used to display the o/p data on the terminal Spain in the user specified format.

⇒ Declaration of printf fun^c return type is int type

```
int printf (const char * src, ...);
```

... ⇒ we can pass single input or multiple inputs.

⇒ printf Fun^c first input is always a string constant. This string constant is a combination of the string which you want to display as it is on the screen + format specifiers + field width + separators + executable characters.

- printf ("%d\n", x);
  - "%d\n" → format specifier
  - \n → executable character
  - , → separators
  - x → i/p's
- printf ("welcome");
  - "welcome" → string constant.

⇒ From second input onwards it may be a variable, address (&x) constants (integers, strings etc), expressions (a+b, a-b; ...) or a fun^c also.

⇒ The no. of format specifiers mentioned in the no. of i/p's passed from second i/p onwards.

⇒ To separate the outputs we can use separators (,.-- etc) or executable characters (\n, \t \b \r etc)

⇒ separator can be any symbol or character.

⇒ printf ("%d - %x - %o", x, x, x);

⇒ Printf fun^c evaluates from right to left but prints from left to right.

⇒ printf fun^c will return the count of number of characters displayed on the terminal space.

⇒ printf ("welcome\n");
1 2 3 4 5 6 7 8 (8)

⇒ return value = 8 (length).

⇒ printf fun^c will stop printing the data when it finds null character.

→ printf ("wel\0come");

o/p ⇒ wel
1 2 3

return value = 3

⇒ printf ("%d", size of ("welcome"));

⇒ bytes = 10 will allocate

⇒ we are passing the base address indirectly in printf fun^c & memory is allocated in rodata.

⇒ printf ("%p\n", "welcome"); ⇒ displays base address

⇒ Here we are printing the string's starting base address The memory for this string is allocated in rodata.

⇒ char str[10] = "welcome";

printf (str); o/p = welcome.

↳ string's address

⇒ To the printf fun^c, the first input it can be either string constant or string's base address.

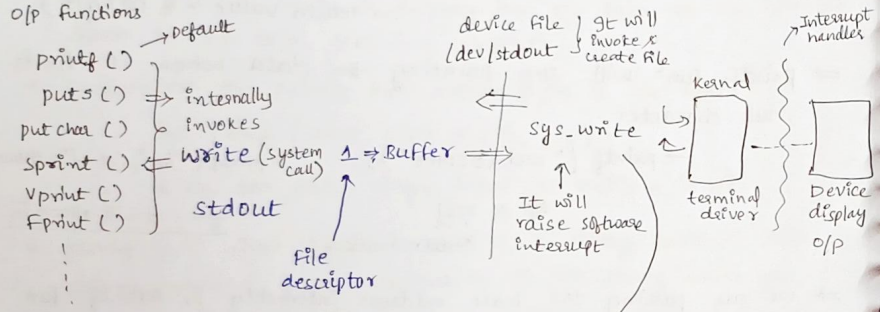
⇒ "ABC" = size 4 bytes, length 3
A B C \0
1 2 3 4

⇒ "welcome\n" = 9 bytes (8)
1 2 3 4 5 6 7 8 9

⇒ "wel\0come\n" = 10 bytes (3)
1 2 3 4 5 6 7 8 9 10

⇒ printf fun^c internally invokes write system call, from there it is sent to buffer from that it will raise software interrupt sys_write. It flows to terminal driver & reaches to device.

⇒ standard 'c' library o/p functions



⇒ %c ⇒ 1 byte ⇒ Binary to ASCII value ⇒ ASCII character.

%d ⇒ 4 bytes ⇒ Binary to signed decimal

%u ⇒ 4 bytes ⇒ Binary to unsigned decimal

%i ⇒ 4 bytes ⇒ Binary to decimal.

%x (or) %x ⇒ 4 bytes ⇒ Binary to Hexa decimal (4 bits = 1 hexa digit)

%o (or) %o ⇒ 4 bytes ⇒ Binary to octal (3 bits = 1 octa digit)

%f (or) %f ⇒ 4 bytes ⇒ Binary to float.

%E (or) %e ⇒ 4 bytes ⇒ Binary to exponential

%g (or) %g ⇒ 4 bytes ⇒ Binary to float (or) exponential (total of 6 digits)

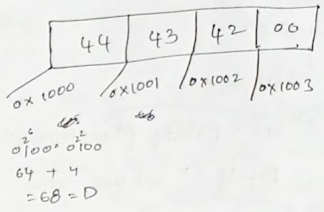
%s ⇒ starting string address ⇒ Byte by byte until it find \0.

%p ⇒ address ⇒ address in Hexa decimal with 0x.

⇒ unsigned int x = 0x00424344;

printf (%x);

o/p 0xD (address)



* Executable characters :-

⇒ Executable characters are non printable characters they simply changes the cursor position. These executable characters along with field width, we can display the output in a particular format (any user defined format).

① "\n" → New line character which changes the cursor position from current line to next line starting position.

⇒ New line character represents the enter key from the keyboard.

⇒ char x = "\n";

printf ("%c", x); ⇒ Empty space jumps to new line.

printf ("%c %d", x, x); ⇒ 10 (Ascii value of \n)

② "\b" → Back space :- (cursor will jump one previous position)

③ "\t" → Horizontal tab (Eight empty spaces)

④ "\v" → Vertical tab (cursor will jump from current position to next line same position)

⑤ "\f" → Form Feed (This is mainly designed for printers cursor will jumps to next logical page)

⑥ "\r" → carriage return (It is designed for the command to change the cursor from current position to same line starting position)

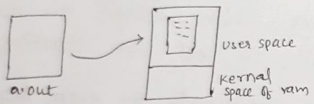
⑦ "\a" → Alert (It will generate a bell sound).

⇒ test.c

```
main()
{
    int ret;
    ret = printf("welcome");
    printf("return value is %d\n", ret);
    ret = printf("Hi \t \r Hello \r Hyderabad \n");
    printf("output \t %d\n", ret);
}
```

⇒ Pgm will be inserted & run in the compiler

$ gcc test.c



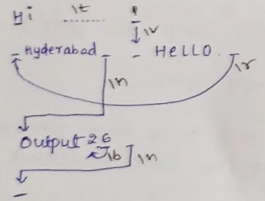
Output :-

$ a/a.out :-

when we click on enter

welcome return value is 7 \n

Hi \t \r



⇒ printf("%d", 123); ⇒ return value = (3)

⇒ printf("%f", 123.23) ⇒ 123.230000 (1 2 3 4 5 6 7 8 9 10) ⇒ ret value = (10)

⇒ printf("%d\n", printf("welcome\n")); (1 2 3 4 5 6 7 8) ⇒ welcome \n, 8 \n

✱ First priority is always given to the inner func.

✱ Field width :-

⇒ It will reserve the no. of empty spaces for the given i/p.

✱ The output will not change.

⇒ The field width can be positive number or negative number or even it can be a real constant. It must be used along with the format specifier. so positive field width will justify the output towards right hand side. The Negative field width will justify the o/p towards left hand side.

⇒ printf("%10d\n", 123); ⇒ _ _ _ _ _ _ _ 1 2 3 \n

⇒ return value = (11)

⇒ printf("%10d\n", 12345); ⇒ _ _ _ _ _ 1 2 3 4 5 \n ret value = (11)

⇒ printf("%-10d %-10d\n", 123, 12345);

1 2 3 _ _ _ _ _ _ _ 1 2 3 4 5 _ _ _ _ _ \n

ret value = (23) including space & \n.

⇒ printf("%2d\n", 12345); ⇒ 1 2 3 4 5 \n (output will not change)

⇒ Real constants

printf("%f", 123.25); ⇒ 123.250000.

⇒ printf("%10.2f", 123.25); ⇒ _ _ _ _ 1 2 3 . 2 5

(10 → spaces, 2 → digits after decimal point)

⇒ printf("%10.2f", 1234.54f67); ⇒ _ _ _ 1 2 3 4 . 5 4

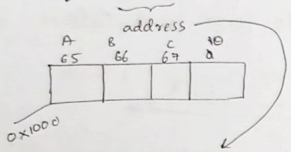
⇒ printf("%-10.2f", 1234.54672); ⇒ 1 2 3 4 . 5 4 _ _ _

02/10/2022

⇒ In 64 bit machine pointer size is 8 bytes.

⇒ In 32 bit machine the address length is always 32 bit i.e., size is 4 bytes (address size & pointer size is same).

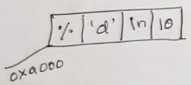
① printf ("ABC");



printf (const char * src, ...);

⇒ It goes to rodata

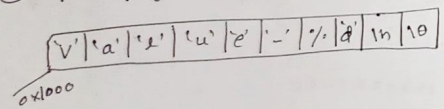
② printf ("%d\n", 12);

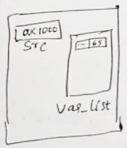


→ goes to rodata

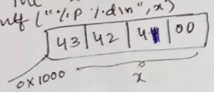
⇒ 12 \n

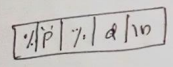
③ printf ("value - %d\n", A);



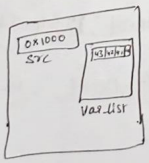


④ unsigned int x = 0x00414243;
printf ("%p %d\n", x)





→ goes to rodata



⇒ Field width is used in string constants.

⇒ return value (11)
- printf ("%-10s\n", "welcome"); ⇒ welcome---
⇒ printf ("%10s\n", "welcome"); ⇒ ---welcome
⇒ printf ("%-10.3s\n", "welcome"); ⇒ wel-------
⇒ printf ("%10.3s\n", "welcome"); ⇒ -------wel

(*) NOTE :-

* put char() is used to print only single integer.
* put s() is used to print string constants. we can't use to print integers.

⇒ char x = 'abc';
printf ("%c", x);

ⓐ a
ⓑ compilation error
ⓒ c
ⓓ None

⇒ Little Endian = It prints c.
⇒ Big Endian = It prints a

(#) Scanf function :-

⇒ It is a standard c library input function.
⇒ Generally, it behaves as a blocking call that means the execution of the program will be suspended until the user enters the input.
⇒ To check the declaration of scanf func
- $ man 3 scanf.
- Format ⇒ int scanf (const char * src, ...);

⇒ scanf ("%d", x); ⇒ If we give 12 as i/p, it will take
⇒ scanf ("input - %d", &x); ⇒ If we give 12 as i/p, it will not accept. In this case we have to give as input - 12

⇒ scanf fun^c first input it is string constant. The string constant is the combination of format specifiers, field width, separators. It will not be an executable files.

⇒ From second input onwards, we have to pass memory locations addresses only.

⇒ so scanf always deals with memory locations addresses only.

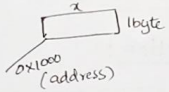
⇒ Number of format specifiers are mentioned in first i/p should be equal to the no. of memory location addresses from second i/p onwards.

⊛ Reading a character:-

⇒ %c format specifier is used to read the single character. we cannot use any field width (No field width is applicable for %c format specifier)

⊙
```
main()
{
char x;
scanf("%c", &x);
```

⇒ It will allocate 1 byte memory



⇒ By using scanf fun^c, use one printf stmt to display what type of input to be entered.

⊙
```
main()
{
char x;
printf("enter a character\n");
scanf("%c", &x);
```

o/p
Enter a character,
1

ASCII 1 = 49 = 0x37
%d = 49
%c = 1

⇒ program execution is suspended until you provide the input.

⇒ As soon as you provide the input it pass to address location.

⇒
```
main()
{
char x;
printf("enter 3 characters");
scanf("%c %c %c", &x, &x, &x);
printf("%c %d\n", x, x);
}
```

o/p
Enter 3 characters
abc↵
1byte [x] (c overwrites)

(%c = c
%d = 67)

⊛ scanf fun^c is evaluated from left to right and reads the output from left to right only.

⊛ If we use separators, we must provide the input along with separators.

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

o/p terminal
Enter 3 characters
⇒ abc (It wont read 3 characters but reads only one)
⇒ a b c [It reads 3 characters]
⊛ spaces acts as separators

⇒ when we use separators in scanf fun^c, the input must be provided along with separators in the same format specified in first input.

⇒ when we are reading multiple characters no default separators are used because the length is defined i.e., one (1) character (1 byte)

⇒ But when we are reading multiple integer constants & string constants we have to use either default separators or our own separators.

⇒ Enter key or space key are default separators.

Reading Integer constants :-

⇒ we can use %i format specifier to read integer constant either in decimal, Hexa decimal (or) octal. For Hexa decimal we have to use 0x prefix & for octal 0 prefix.

⇒ To read the input in decimal we have to use %d.
⇒ To read " " " Hexa decimal we " " " %x.
⇒ To " " " " octal we " " " %o.

⇒ when we use our own separators while reading multiple integers we can't use default separators.

⇒
```
main()
{
  int x;
  printf("Enter integer constant");
  scanf("%d", &x);
  printf("%d\n", x);
}
```
10 ←

→ o/p = 10.

⇒
```
main()
{ int x,y,z;
  printf("Enter 3 integers\n");
  scanf("%d %d %d", &x, &y, &z);
  printf("%d-%d-%d\n", x,y,z);
}
```
10 20 30

⇒ 10↵ 20↵ 30

10↵ 20↵ 30

⇒ Scanf Func will not comes out until you provide 3 inputs when we use default separators.

⇒ When we use field width in scanf func that will effect the input.

```
int x;
scanf("%3d", &x);
```

↳ Here, the i/p will read up to 3 digits only.
↳ Here we can read maximum up to 3 digit value cuz of field width

↖ 12345
It will read only 123

↖ 12
It will read 12.

⇒
```
int dt, mnt, year;
printf("dd-mm-yyyy\n");
scanf("%2d %2d %4d", &dt, &mnt, &year);
```

① ↖ 01-10-2022

② 01↵ 10↵ 2022↵

③ 01 10 2022
dt ↑ mnt ↑ year

④ 11 02 0 2022
dt ↑ mnt ↑ year ↑ neglects this

⑤ 1 10 2 2022
dt ↓ mnt ↓ year ↘

✱ Reading string constants :-

⇒ To read the string constants we have to use %s format specifier.

⇒ we can request the memory for a string constant by declaring a character array.

< Qualifier > < sign Qualifier > < Modifier > < data type > < array name > [n] ;
n ↑ No. of elements.

⇒ "ABC" ⇒ 3 characters + 1 null character.
so we need 4 bytes

① char str[4];

⇒ Here, we are requesting 4 bytes memory to store a string constant. In this 4 bytes memory, it will store corresponding ASCII character ASCII values in binary format.

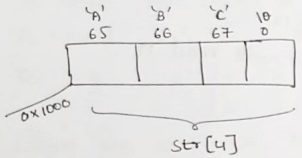
⇒ we can initialize a character array in 2 ways:-

① we can directly mention in a pair of double quotes "ABC"

- char str[4] = "ABC";

② we can assign individual characters in pair of curly braces.

- char str[4] = {'A','B','C','\0'};



⇒ Data stores in Binary format.

⇒ Array name it will return array starting base address.

⇒ printf("Enter a string constant (4)");
scanf("%s", str);

↖ RAM

R	A	M	\0

⇒ when we are reading string constants by using scanf func^c the null character gets updated at the end of the string by scanf func^c.

⇒ In a file we dont store null character (\0). It is used only for the purpose of string constant termination.

⇒
```
main()
{
    char str[20];
    scanf("%s", &str);
    printf("%s", str);
}
```

↖ kumar gampala

⇒ It will print kumar only cuz space (separator) acts as termination.

- scanf("%[^\n]s", str);

⇒ It will print kumar gampala, cuz this format specifier will scan separators also.

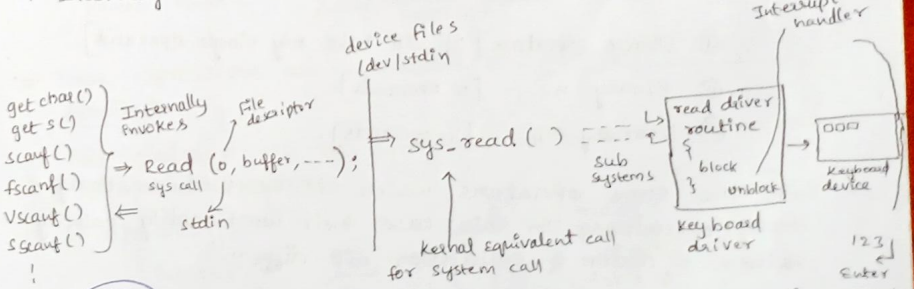
⇒ If we want to read the i/p as a stmt or as a sentence we can use gets(str);

⇒ It is dangerous to use as it continuously prints & overwrites the content of memory location (out of boundaries memory).

⇒
```
main()
{
    int x;
    char y;
    printf("Enter integer constant");
    scanf("%d", &x);
    printf("Enter character");
    scanf("%c", &y);
}
```

⇒ In this program we have 2 scanf functions the program execution should be suspended 2 times but practically it will suspended 2 times but practically it will suspend only one time. This is the draw back by using scanf function. To overcome this we have to use a func^c --fpurge(stdin); immediately after scanf func^c.
(stdin → standard input file descriptor)

⇒ Internally it will reset the standard input buffer.



NOTE :-

. In /dev directory consists of 3 device files

① stdin → used for reading i/p's → It is linked to keyboard driver

② stdout → used for displaying o/p's.

③ stderror → used for displaying error msgs. } It is linked to terminal driver.

⇒ 03/10/2022

(*) Operators :-

⇒ Operators are used to write expressions. Expressions are also 'c' stmts.

⇒ we can write simple expressions (or) complex expressions.

⇒ Expressions are evaluated by the compiler based on parse tree method & for each operation it will generate assembly instruction based on priority values & order of evaluation.

```
x = 10
    ↑
  ① operation

x = (5+3)/2 - 10;
      ↑ ↑ ↑ ↑
      ④ ① ② ③
   4 operations
```

⇒ These operators will have special meaning in 'c' programming language, because of this reason we can't use them in between identifiers.

⇒ Compilers knows the meaning of each & every operation.

⇒ These operators are classified into 3 types based on operand. Operand can be a variable or constant. The operators are :-

① Unary operator [It will take only single operand]
② Binary " [2 operands]
③ Ternary " [3 operands].

⇒ There are some operators which can be used as Unary & binary operators. In this case their functionality, priority values & order of evaluation will differs.

⇒ Based on functionalities, these operators are classified as below :-

① Arithmetic operators
② Increment "
③ Decrement "
④ Assignment operators
⑤ Bitwise " (~)
⑥ Relational "
⑦ conditional "
⑧ Logical " (!)
⑨ Size of "
⑩ Miscellaneous "
⑪ comma "

(*) Assignment operators :-

⇒ Its priority value is 14.
⇒ order of evaluation is from Right to left.
⇒ It comes under Binary operator.
⇒ This operator is used to assign a value to a particular memory location.

(assigns the values to memory location)

④ Memory location = constants / variable name / address / expression / function

variable name (or) pointer variable

Assignment operator

⇒ These expressions are evaluated in second (2nd) stage of compilation.

⇒ This operator can be combined with arithmetic operators & Bitwise operators

```
⊙ main()
  { int a=5;
    int b=10;
    a=b;
    b=a;
    printf("%d-%d\n", a, b);
  }
```

a=5, b=10 → a=b → a=10, b=10 → b=a → a=10, b=10

o/p = 10-10

```
⊙ main()
  { int a=5;
    int b=10;
    printf("%d-%d\n", a=b, b=a);
  }
```

o/p = 5-5

a=5, b=5

✳ write a program to swap the content of two variables by using third variable.

⇒
```
main()
{
 int a,b,temp;
  a=5;
  b=10;
  temp=a;
  a=b;
  b=temp;
printf("%d-%d-%d\n",a,b,temp);
}
```

O/P

a=5 b=10

temp = a ⑤

a = b ⑩

b = temp ⑤

temp=5, a=10, b=5

✳ Size of operators :-

⇒ It is built-in operators in compilers.

⇒ It will return the size of number of bytes decided by the compiler for the variable / constant (or) data type.

⇒ Its precedence (priority value is 2).

⇒ Order of evaluation is from right to left.

6/10/2022

✳ Miscellaneous Operators :-

⇒ ( ) = paranthesis (for expressions), [ ] = square brackets (For arrays), → arrows (By pointer name)

. = dot (To access members by variable name)

⇒ Least value will have highest priority.

⇒ Highest value will have least priority.

⇒ priority value is 1.

⇒ order of evaluation is from left to right.

⇒ Some operators can be used as unary operator as well as binary operator.

	used as	Unary operator	&	Binary operator
*	→	pointer (2)	—	Multiplication (3)
&	→	Address (2)	—	Bitwise AND operation (8)
+	→	Represents +ve value (2)	—	addition (4)
-	→	Represents -ve value (2)	→	Subtraction (4)

order of evaluation is from Right to left

order of evaluation is from left to right.

✳ comma Operator :- (15) → Highest value, least priority.

⇒ order of evaluation is from left to right.

⇒ This operator is used to separate the inputs. you can combine multiple 'c' statements with comma (,) operator.

⊙
```
main()
{
 int x;
 int y;
 int z;
}
```
⇒ It can also be written as `int x,y,z;`

⊙
```
main()
{
 int x=5;
 int y=20;
 int z;
 z=x, x+y, x+y+20, x=20;
printf("%d-%d-%d\n",x,y,z);
}
```

O/P

x = 5 → 20

y = 20 → y=20 ✓

z = x ; → z=x = z=5 ✓

x+y ; → 5+20=25

x+y+20 ; = 5+20+20

x = 20; → x=20 ✓

O/P = 20-20-5.

⇒
```
main()
{
  int x=5;
  int y=20;
  int z;
  z=(x,y=50,z=40,x+y+z);
  printf("%d-%d-%d\n",x,y,z);
}
```

```
x;
y=50;
z=40;
z=x+y+z   (It evaluates right most one when we use comma operator)
50+50+40 = 90
```

O/P = x=5
y=50
z=90

⇒
```
main()
{
  int x=5;
  x=+2, x+=2, x=+x;
  x+=x;
}
```
x=?

x = 5 +2 +2 +4 = 8

x=+2 , x+=2 ⇓ x=x+2 = +2+2 x=4

x=+x x=+4 x+=x x=x+x 4+4 =8

O/P
8

⇒
```
main()
{
  int x=5;
  x++;
}
```
x=?

x = 5 → 6
x=5
x=6

⇒
```
main()
{
  int x=5;
  ++x;
}
```
x=?

x=5
++x (First increments then assigns the value)
x = 6

⇒
```
main()
{
  int x=5;
  int y;
  y=x++;
}
```
x=?, y=?

y=x++ ⑤ → x=x+1 = 5+1 x=6
y=5

x=6, y=5

⇒
```
main()
{
  int x=5;
  int y;
  y=++x;
}
```
x=?, y=?

y=++x ; → y=++x ; ⇓ 5+1 = 6

y=6, x=6

⇒ y=x++ (+) x++ ;

- These type of expressions outputs varies from compiler to compiler
- In Real time development we are not going to use these type of expressions.

(*) Increment & Decrement Operators :-

++ } (2) , order of evaluation is from
-- } Right to left.

⇒ These operators are not applicable on constants.
⇒ These operators are applicable only on variables.
⇒ These are again classified into 2 types :-
① pre increment ⇒ ++<variable name>.
② post increment ⇒ <variable name>++.
⇒ This increment operator is used to increment the variable value by 1.
⇒ In pre increment, First increments that variable value by '1' & assigns to that variable.
⇒ In post increment, it will first assigns the value & then increments the variable value by '1'.

* Pre-decrement :-

--<variable name>; (First decrements & then assigns the value.)

* post decrement :-

<variable name>--; (First assigns the value & then it decrements)

⇒
```
main()
{
  int x=5;
  x--;
}
```
x=?

O/P x = 5 4 ; x-- ; 5-1=4 x=4

⇒
```
main()
{
  int x=5;
  --x;
}
```
x=?

O/P x = 5 4 ; --x 5-1=4

⇒
```
main()
{
  int x=5;
  int y;
  y=x--;
}
```
x=?, y=?

O/P y=x-- ; ⑤ ↓ x=x-1 5-1=4 y=5 x=4

⇒
```
main()
{
  int x=5;
  int y;
  y=--x;
}
```
x=?, y=?

O/P y=--x ; 5-1=4 x=4

y=4 x=4

```
main()
{
int x=5;
printf("%d\n", x++);
x=5;
printf("%d\n", ++x);
x=5;
printf("%d-%d\n", x++, x++);
x=5;
printf("%d-%d\n", ++x, ++x);
x=5;
printf("%d-%d\n", ++x, x++);
x=5;
printf("%d-%d\n", x++, ++x);
x=5;
printf("%d-%d-%d\n", x++, ++x, x++);
x=5;
printf("%d-%d-%d-%d\n", ++x, ++x, x++, ++x);
}
```

To understand printf func^{c} clearly

++x will checks but not substitutes at that instant it will verify all other stmts

```
⇒ main()
{ int a=5
  int b=10;
  a=b;
  b=a;
printf("%d-%d\n", a, b);
}
```

o/p
a = 5
b = 10
a = b$_{10}$
b = a$_{10}$

a=10, b=10

```
⇒ main()
{
int x=5;
x=x+1;
}
```
x=6

```
main
{
int x=5;
x++;
}
```
x=6

(same operation will performed we can write in any one of the above format)

(*) gcc -save-temps <file name.c>

.i
.s → see how many instructions are executed (machine instructions)
.o
a.out

(*) Arithmetic Operators :-

⇒ To perform mathematical arithmetic operations like addition, subtraction, multiplication, divison etc we use these below set of operators.

⇒ It requires 2 operands, so, it comes under binary operator

+, - } 4
*, /, % } 3

order of evaluation is from left to right.

⇒ % (modulus operator) will return the remainder after divison.

⇒ we can't use Real constants with modulus (%) operator.

(*) write a 'c' program to swap 2 no's without using third variable.

⇒
```
main()
{
int a=1;
int b=6;
a=a+b;
b=a-b;
a=a-b;
}
```

a = 1+6 = 7
b = 7-6 = 1
a = 7-1 = 6
⇒ a=6, b=1 ✓

(or)

```
int a=1;
int b=6;
b=a+b;
a=b-a;
b=b-a;
```

(or)

```
main()
{
int a=2;
int b=3;
b=a*b;
a=b/a;
b=b/a;
```

(or)

```
int a=2;
int b=3;
a=a*b;
b=a/b;
a=a/b;
```

a = a*b = 2*3 = 6
b = a/b = 6/3 = 2
a = a/b = 6/2 = 3
a = 3, b = 2 ✓

* Type Conversions :-

① Implicit type conversions
② Explicit type conversions.

① ⇒ The conversion is done automatically by the compiler.

② ⇒ In Explicit type conversion, it is specified by the user by type casting.

⇒ During the implicit type of conversion, compiler automatically follows the order.

⇒ All arithmetic operators comes under binary operators.

⇒ It requires 2 operands.

⇒ By default, real constants comes under double precision type.

long double
↓
double
↓
float
↓
long int
↓
int
↓
short int
↓
char.

① printf("%d\n", 2 + 3.5);
int → converts to higher order (i.e double)
double
This conversion is known as Implicit conversion.

double = 5.500000000

o/p = garbage value

use "%f" to get o/p = 5.500000000.

② printf("%d\n", 2 + (int) 3.5);
int
double
converting double type into int type.
2+3=5
This type of conversion is known as type casting (Explicit conversion)

⇒ when we are type casting a value (or) variable, the modification is substituted at that instant only, that will not be applicable throughout the program.

③ printf("%f", 2+3); 5.000000 But it prints garbage value

④ printf("%f", 2 + (float) 3); ⇒ 5.000000.

⑤ printf("%f", (float) 2 + 3); ⇒ 5.000000

```
⇒ main()
  {
    unsigned int x = 0XAABBCCDD;
    unsigned int y;   | int y=0;
      y = (char) x
    printf("%x\n", y);  ⟶ o/p = 0XFFFFFFDD (garbage value)
      y = x
    printf("%x\n", y);  ⟶ o/p = 0XAABBCCDD.
  }
```

```
⇒ main()
  {
    float y;
      y = 2+3.5;
    printf("%f\n", y);   ⇒ o/p 5.500000.
  }
```

07/10/22

⇒ printf ("%d\n", 5/2); ⇒ 2

⇒ printf ("%f\n", 5.0/2); ⇒ 2.500000.

(5.0 → double, 2 → int → double) } Implicit conversion.

⇒ printf ("%d", 5.0/2); ⇒ garbage value. (Answer is in double type but the format specifier is in decimal type).

(5.0 → double, 2 → int → double) } Implicit

⊛ Relational Operators :-

< , > , <= , >= } 6

!= , == } 7

} Order of evaluation is from left to right.

⇒ These operators are used to know the relation between 2 operands

⇒ In general we use to write the conditions to compare.

⇒ These operators will return boolean values i.e, True or False

- True is always a Non-zero value (other than zero) (1).
- False is always zero (0).

⇒ printf ("%d", 5>1); ⇒ It prints 1. (True = 1)

⇒ printf ("%d", 5<1); ⇒ It prints 0. (False = 0)

⇒ printf ("%d", 5==1); ⇒ 0 (False)

⇒ printf ("%d", 5!=1); ⇒ 1 (True)

⇒ printf ("%d", 5>=1); ⇒ 1 (True)

⇒ printf ("%d", 5<=1); ⇒ 0 (False)

⇒ x>=10 = Range {10, 11, 12 ---}

x≥10 = {11, 12, 13 ---}

x != 0 ⇒

x	x
-1	1
-1 = 0 (T)	1 != 0 (T)

x > 0 ⇒

x	x
-1	1
-1 > 0 (F)	1 > 0 (T)

⊛ Conditional Operators :-

Syntax :-

condition ? expression 1 : expression 2 ;

⇒ conditional operator is the alternative of if else statement

	condition ? exp 1 : exp 2
Relational operators ←	(T) ✓ ×
Relational with logical ←	(F) × ✓
Relational with Bitwise ←	
Relational with Bitwise & logical ←	

⇒ If the condition is evaluated to True, it will evaluate expression 1 & it will skip expression 2.

⇒ If the cond^n is evaluated to false, it will skip exp 1 & it will evaluates exp 2.

(*) Write a c program to find out biggest among 2 numbers

⇒
```
main()
{
  int x;
  int y;
  int big;
  Printf("enter 2 i/p's x:y\n");
  scanf("%d:%d", &x, &y);
```

(I) `big = (x>y) ? x : y ;`

10>20 → F, x, 20

20>10 → T, ✓ 20, x

⇒ 10:20 (x, y)

⇒ 20:10 (x, y)

(II) `(x>y) ? printf("%d\n", x) : printf("%d\n", y);`

(*) Based on condition we have to write exp 1 & exp 2.

→ we have to write true condition's expression first & then False condition's expression next.

(III) `(x<y) ? big = y ; big = x ;`

(OR)

`big = (x<y) ? y : x ;`

10<20 → T, ✓, x

⇒ whenever we write the cond" it should satisfy in all the possible scenario's (cases).

⇒

x	y	z	
5	3	2	6 possible cases
5	2	3	
2	5	3	
2	3	5	
3	2	5	
3	5	2	
5	5	5	odd cases
5	5	3	

⇒ For this we have to use nested conditional operators.

`big = (x>y) ? ((x>z) ? x : z) : ((y>z) ? y : z);`

5>3 (T), 5>2 (T), 5, x (skip), x (skip)

`big = 2>5 (F) : ((5>3) ? y : z)` → x (skip); T → 5, x (skip)

(*) write a c program to verify the given i/p is divisible by '5' or not.

Logic: x%5==0

⇒
```
main()
{
  int x;
  printf("enter a value\n");
  scanf("%d", &x);
  (x%5==0) ? printf("divisible by 5") : printf("Not divisible by 5");
```

comparing x%5 with zero

```
  (x%5) ? printf("not divisible by 5\n") : printf("divisible by 5");
```

1st statement (F) ← 0

(*) Logical Operators :-

&&	=	logical AND
||	=	" OR
!	=	" NOT

⇒ whenever we want to combine two conditions (or) expressions we have to use logical operators.

⇒ It must satisfy both the conditions they are combined with logical AND operators.

⇒ It must satisfy anyone of the cond", they are combined with logical OR operator.

⇒ If we want to change the entire expression from True to False or from false to true we have to use logical NOT operator.

✳ Write a c prgm to verify the given value lies b/w 9 to 99.

⇒
```
main()
{
int x;
PF("Enter the i/p\n");
SF("%d", &x);
((x>=9)&&(x<=99))?PF("x is within the range"):PF("x is out of range");
((x>8)&&(x<100))?PF("x is within the range"):PF("  "  ");
```

10 ✓ 10 ×

⇒ using logical AND

10>=9 (T) 10<=99 (T) → (T)

⇒ using logical OR

```
((x<9)||(x>99))?PF("x is out of range\n"):PF("x is within range");
```

10<9 (F) 10>99 (F) → (F)

× ✓

⇒ By considering within the range use logical AND (&&)

⇒ By considering out of range use logical OR (||).

⇒ When we use logical operators, the expressions must be written with relational operators.

exp1	&&	exp 2	o/p
T		T	T
T		F	F
F		T	F
F		F	F

exp 1	||	exp 2	o/p
T		T	T
T		F	T
F		T	T
F		F	F

! (exp 1)	o/p
! (T)	F
! (F)	T

⇒ If we want to compare more than 2 expressions they must be combined with logical AND (or) logical OR.

✳ Write a c prgm to verify the given value is divisible by 3 & 5 using logical AND and logical OR.

⇒
```
main()
{
int x;
printf("Enter i/p");
scanf("%d",&x);
((x%3==0)&&((x%5)==0))?PF("x is divisible by 3 & 5"):PF("x is not divisible by 3 & 5");
((x%3!=0)||(x%5!=0))?PF("x is not divisible by 3 & 5"):PF("x is divisible by 3 & 5");
```

15 ✓ 15 ×

15%3==0 (T) 15%5==0 (T) → (T)

15%3!=0 (F) 15%5!=0 (F) → (F)

✳ Write a c program to verify the given character is upper case or lower case.

⇒
```
main()
{
char x;
PF("Enter a character");
SF("%c", &x);
```

A ---- Z
65 90

a ---- z
97 122

0 ---- 9
48 57

⇒ When we are comparing ASCII characters, internally it will compare ASCII values.

```
((x>=65)&&(x<=90))?PF("uppercase"):(((x>=97)&&(x<=122))?PF("lower case")
: PF("other than alphabet"));
```

(or)

```
((x>='A')&&(x<='Z'))?PF("upper case"):(((x>='a')&&(x<='z'))?
PF("lower case"):PF("other than alphabet"));
```

(*) write a C prgm to check the given character is vowel or not

⇒

```
main()
{
  char x;
  PF("Enter the i/p\n");
  SF("%c", &x);
((x=='A')||(x=='E')||(x=='I')||(x=='O')||(x=='U')||(x=='a')
 ||(x=='e')||(x=='i')||(x=='o')||(x=='u'))?PF("vowel"):
                                  PF("NOT a vowel");
```

A, E, I, O, U
a, e, i, o, u.

⇒

```
main()
{
  int x=1;
  int y=0;
((x>0)&&(y++))?PF("True"):PF("False");
```

x>0 : T, F ; y : 1, 0 ; (F)

x = ?, y = ? x = 1, y = 1

⇒

```
main()
{
  int x=0;
  (x++)?PF("T"):PF("F");
  PF("%d\n", x);
}
```

(x++) : 0 → F ; x : 0 → 1 ; False ; x = 1

⇒

```
main()
{
  int x=1;
  int y=0;
((x>0)||(y++))?PF("T"):PF("F");
```

x>0 : T ; y : 0 → F

x = 1, y = 0

⇒ when we use logical operator, compiler will perform optimization.

exp1 && exp2 → o/p

exp1	exp2	o/p
F	F	F
F	T	F
T	F	F
T	T	T

It will not evaluate if first exp is F

⇒ directly o/p is False, if one exp is false

exp1 || exp2 → o/p

exp1	exp2	o/p
T	T	T
T	F	T
F	T	T
F	F	F

⇒ an OR operation if first exp is true it will not evaluate other stmts, directly o/p is true.

In this 2 cases, compiler will perform optimization technique. 2nd expression is not evaluated by the compiler

⇒

```
main()
{
  int x=1;
((x>0)&&(printf("welcome")))?PF("T"):PF("F");
```

x>0 : T ; printf : 7 ; x = ?, y = ?

↳ x = 1, y = 7.

⊙ 7 is a non-zero value. ∴ It considers as true stmt.

⊙ Func^c call (or) Func^c invocation after performing the task it is replaced with the return values.

⇒ Block statements :- Group of 'c' stmts are combined with pair of curly braces that represents block stmts.

⇒ For compiler point of view it will consider as a singe stmt.

(*) NOTE :-

⇒ semi colon & pair of curly braces are not allowed in conditional operator. In the place of semicolon we can use comma operator and in the place of pair of curly braces we can use paranthesis.

```
→ main()
{ int x=10, y=20; big;
(x>y)?(PF("biggest value is \n")), PF("%d\n",x):PF("Biggest
                                    value is \n")), PF("%d\n",y);
```

pair of curly braces replaced with paranthesis

semi colon replaced with comma

08/10/22

✳ Bit Wise Operators :-

⇒ To manipulate the data at bit level, we have to use Bit wise operators. we can manipulate the data with the help of variables & pointers.

⇒ A communication based device will have memory in terms of Hardware registers

⇒ protocols → user manual

⇒ Any communication based device will have 3 set of registers

1. Data registers
2. control "
3. status "

① ⇒ The Data registers acts as buffer for exchanging the data between devices. On these data registers we can have both read & write permissions.

⇒ we can access the memory by register name.

② ⇒ Based on our requirement we can send (or) write operations to set communication parameters to correct error we use control registers.

⇒ On control registers we have both read & write permissions

I2C ⇄ I2C ✓

I2C ⇄ UART ✗

UART ⇄ UART ✓

③ Status Registers :-

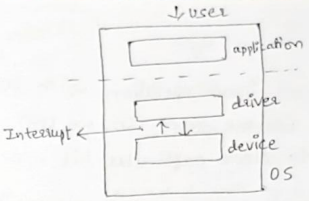
→ A device will update its status in to status registers.

⇒ status are succesfully sent, successfully received, errors.

⇒ Device will have write permissions.

⇒ User will have only read permissions.

⇒ To control these H/w registers, manipulate the data. we have to manipulate with Bit wise operators.



Interrupt handler fun°.

⇒ when we use bitwise operators, the variable type must be unsigned type.

⇒ This set of Hardware registers can be 8 bit, 16 bit or 32 bit.

8 bit → unsigned char

16 bit → unsigned short int

32 bit → unsigned int.

⇒ Basic bit level Manipulations :-

Set a bit → 0/1 ⇒ 1

Reset / clear a bit → 0/1 ⇒ 0

Status of a bit → { 0 → 0, 1 → 1

Toggle a bit → { 0 → 1, 1 → 0

* Bitwise operators :-

1. Bitwise AND (&)
2. Bitwise OR (|)
3. Bitwise XOR (^)
4. Negation (~)
5. Left shift (<<)
6. Right shift (>>)

① Bitwise AND (&)

bit x	bit y	x & y
0	0	0
1	0	0
0	1	0
1	1	1

To clear ⇒ rows 1 & 2

To know status ⇒ rows 3 & 4

⇒ If we perform AND operation with zero the o/p will become zero. So, we can use this to clear particular bit.

⇒ If we perform AND operation with 1 the output will be same as input. So, we can find the status of the bit.

⇒ AND operation with zero ⇒ clears the bit.

⇒ AND operation with one ⇒ status of the bit.

② Bitwise OR (|) :-

bit x	bit y	x | y
0	0	0
0	1	1
1	1	1
1	0	1

To set the bit ⇒ rows 2, 3 & 4

⇒ Bit wise OR with one we can set a particular bit.

③ Negation (~) :-

Bit x	~x
0	1
1	0

⇒ used to toggle the entire o/p.

④ Bitwise X-OR (^) :-

Bit x	Bit y	x ^ y
0	0	0
0	1	1
1	1	0
1	0	1

⇒ To toggle the particular bit.

10/10/22

1 Byte ⇒ 0^{th} – 7^{th} bit.

2 bytes ⇒ 0^{th} – 15^{th} bit

4 bytes ⇒ 0^{th} – 31^{st} bit.

0^{th} bit → 0000 0001 ⇒ $2^0 = 1$ ⇒ 0x01 ⇒ 0x1<<0

1^{st} bit → 0000 0010 ⇒ $2^1 = 2$ ⇒ 0x02 ⇒ 0x1<<1

2^{nd} bit → 0000 0100 ⇒ $2^2 = 4$ ⇒ 0x04 ⇒ 0x1<<2

3^{rd} bit → 0000 1000 ⇒ $2^3 = 8$ ⇒ 0x08 ⇒ 0x1<<3

4^{th} bit → 0001 0000 → $2^4 = 16$ ⇒ 0x10 ⇒ 0x1<<4

5^{th} bit → 0010 0000 → $2^5 = 32$ ⇒ 0x20 ⇒ 0x1<<5

6^{th} bit → 0100 0000 → $2^6 = 64$ ⇒ 0x40 ⇒ 0x1<<6

7^{th} bit → 1000 0000 → $2^7 = 128$ ⇒ 0x80 ⇒ 0x1<<7

8^{th} bit → 0000 0001 0000 0000 → $2^8 = 256$ ⇒ 0x0100 ⇒ 0x1<<8

9^{th} bit → 0000 0010 0000 0000 → 2^9 ⇒ 0x0200

10^{th} bit → 0000 0100 0000 0000 → 2^{10} ⇒ 0x0400.

11^{th} bit → 0x0800

12^{th} bit → 0x1000

13^{th} bit → 0x2000

14^{th} bit → 0x4000

15^{th} bit → 0x8000

To clear (for continuous bits)

single bit = 0x1
2 bits = 0x3
3 bits = 0x7
4 bits = 0xF
5 bits = 0x1F
6 bits = 0x3F
7 bits = 0x7F
8 bits = 0xFF.

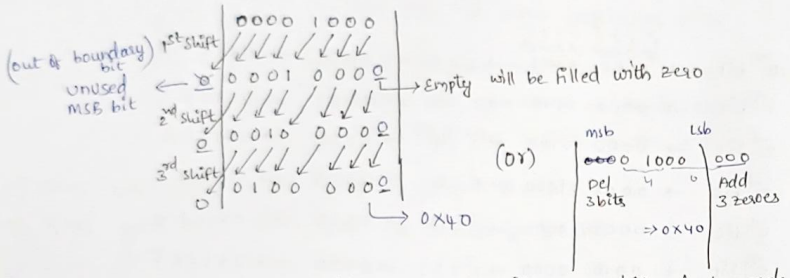
⑤ Left shift (<<) operator :-

Variable (Integer constant) << no. of times bits to be shifted towards left hand side (MSB side).

Eg:- unsigned char x = 0x08;

x = x << 3;

↳ 0x08 left shift 3 times ⇒ 0x08 << 3



⇒ In this left shift operation all the bits are shifted towards MSb side, we can see empty spaces on LSb side they are updated with zero's (0) and the bits which are crossed MSb bit they are ignored.

⇒ For every left shift operation the value is multiplied by '2'. This process should not cross MSb bit (i.e., Maximum size).

⇒ unsigned int x = 0x80;

x = x << 3;

printf ("%x", x)

0000 0000 0000 0000 0000 0000 1000 0000 000

Del 3 bits in MSb — Add 3 zeroes in LSb

↳ 0x00000400 It stores internally | But it prints = 0x400.

⇒ x = 0xFF; clear 5th bit (single bit)

= x = (x & (~(0x1 << 5))); ⇒ logic

⇒ x = 0xFF; clear 2nd & 3rd bit

x = (x & (~(0x3 << 2))); (for continuous 2 bits = 0x3)

⇒ x = 0xFF; clear 2nd 3rd 4th bits

continuous 3 bits = (0x7)

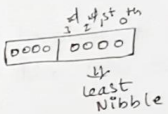
x = (x & (~(0x7) << 2));

⇒ x = 0xFFFFFFFF;

clear 3 bits from 23rd bit

x = (x & (~(0x7) << 23));

⇒ Set least Nibble



x = x | (0xF << 0);

↳ for 4 bits

⇒ 0x00 set 5th, 3rd bit

7 6 5 4 3 2 1 0

0000 0000

(OR) 0010 1000 → 0x28

0010 1000

x = x | 0x28;

x = x | ((0x1 << 3) | (0x1 << 5));

⇒ clear 5th & 3rd bit

x = x & (~((0x1 << 3) | (0x1 << 5)));

eg:- 0xFF

1111 1111

1101 0111 ⇒ ~(0010 1000)

2 8

1101 0111

⇒ Toggle 5th bit

x = x ^ (0x1 << 5);

⇒ Toggle 7th & 3rd bit

x = x ^ ((0x1 << 7) | (0x1 << 3));

⑥ Right shift operator :- (>>)

Variable (Integer constant) >> No. of times bits to be shifted towards Right hand side (i.e., Lsb side).

⇒ If variable is signed type
- If Msb → 0 ⇒ It will add zero's
- If Msb → 1 ⇒ It will add 1's

⇒ If Variable is unsigned type
- ⇒ It will add zeroe's

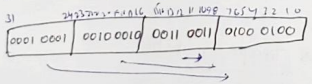
} towards Msb side.

⇒ In this right shift operation all the bits are shifted towards Lsb side, we can see empty spaces towards Msb side. These empty spaces are updated with zero's for unsigned variable. If it is of signed variable the empty spaces are updated depends on Msb bit. If Msb is zero it will add zeroe's, if it is 1 it will add ~~zero~~ one's towards Msb side.

⇒ so the bits which crosses the Lsb bit during the right shift operation they are ignored.

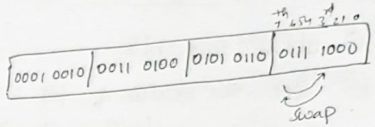
⇒ For every right shift operation the value is divided by 2.

(*) write a c program to extract / print byte by byte from 4 bytes value.

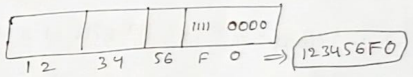


(A) ⇒
```
main()
{
unsigned int x=0X11223344;
unsigned char y;
y=x;
printf("%.x\n", y);          ⇒ 44
y=x>>8;
PF("%.x\n", y);              ⇒ 33
y=x>>16;
PF("%.x\n", y);              ⇒ 22
y=x>>24;
PF("%.x\n", y);              ⇒ 11
PF("%.x\n", x);              ⇒ 11223344
}
```

(*) write a c program to swap 3rd bit with 7th bit for 4 bytes value.



⇒
```
#include<stdio.h>
main()
{
unsigned int x=0X12345678;
unsigned int y, z;
y=((x&(0x1<<3))==0)?0:1;
PF("%.x", y);
z=((x&(0x1<<7))==0)?0:1;
PF("%.x", z);
(y!=z)?((x=x^(0x1<<3)|(0x1<<7)), PF("%.x",x): PF("%.x",x);
}
```



Find status of the bit :-

① Find 3rd bit status

```
unsigned char x=0xFF;
((x&(0x1<<3))==0) ? PF("bit is 0"): PF("bit is 1\n");
```

```
   7 6 5 4 3 2 1 0
   1 1 1 1 1 1 1 1
&  0 0 0 0 1 0 0 0
   0 0 0 0 1 0 0 0
```
→ sample place value ⇒ ∴ Bit status is '1'.

⇒ In status of a bit, we have to verify bit position place value.

② unsigned char x=0xF7;

```
   7 6 5 4 3 2 1 0
   1 1 1 1 0 1 1 1
&  0 0 0 0 1 0 0 0
   0 0 0 0 0 0 0 0
```
→ Not same

∴ Its status is '0'.

11/10/22

* Swap the bits :-

① clear the bits

② Get the status of bit 1 and shift towards bit 2
(status of bit 1) << (bit 2 - bit 1).

③ Get the status of bit 2 & shift towards bit 1.
(status of bit 2) >> (bit 2 - bit 1)

④ Now perform (OR) operation with 1st, 2nd & 3rd operation.

Examples

① x = 0XF0 (7 6 5 4 3 2 1 0 : 1 1 1 1 0 0 0 0)

Swap 7th & 3rd bits

step-1 :- clear 7th & 3rd bits ⇒ 0 1 1 1 0 0 0 0

2 :- Get the status of 3rd bit & shift towards 7th bit

0 0 0 0 0 0 0 0 (7th, 3rd)

(7-3) = 4 times left shift

0 0 0 0 0 0 0 0

3 :- Get the status of bit-2 (i.e. 7th bit) = 1 0 0 0 0 0 0 0
& right shift towards bit 1 (i.e., 3rd bit)
= 0 0 0 0 1 0 0 0

4. perform OR operation

0 1 1 1 0 0 0 0
(|) 0 0 0 0 0 0 0 0
0 0 0 0 1 0 0 0
0 1 1 1 1 0 0 0 ⇒ 0X78

Logic

① ② ③

x = (x & (~(0X1<<7) | (0X1<<3))) | ((x & (0X1<<3))<<(7-3)) | ((x & (0X1<<7))>>(7-3));

(0XF<<28) (0XF<<0)

If swap least nibble with most nibble for 4 bytes

⇒ main()
{
unsigned int x;
unsigned char bit1, bit2, temp;

2nd method use int instead of char
3rd method ⇒ initialize bits before itself.

(use here) To over come overwrite (1st method)

PF ("enter the i/p \n");
SF ("%x", &x);
PF ("enter the bits to be swapped bit 1 : bit 2 (0-31)");
SF ("%d : %d", &bit 1, &bit 2);

((bit 1 >= 0) && (bit 1 <= 31)) ? 0 : (PF ("invalid i/p's \n"), exit (0));

(bit 2 >= 0) && (bit 2 <= 31)) ? 0 : (PF ("Invalid i/p's \n"), exit (0));

(bit 1 > bit 2) ? (temp = bit 1, bit 1 = bit 2, bit 2 = temp) : 0;

x = (x & (~(0X1<<bit 2) | (0X1<<bit 1))) | ((x & (0X1<<bit 2)) << (bit 2 - bit 1))
| ((x & (0X1<<bit 2)) >> (bit 2 - bit 1));

NOTE :-
⇒ exit (0) func will terminate the program there itself.

⇒ swap 1st byte with last byte x = 0X 1122 3344

31	23	16	15	8	7	0
0001 0001	0010 0010	0011 0011	0100 0100			

→ main()
{
unsigned int x;
unsigned char byte 1, byte 2;
printf ("Enter the i/p \n");
scanf ("%x", &x);
printf ("Enter the bytes to be swapped byte 1 : byte 2");
scanf ("%x : %x", &byte 1, &byte 2);

Clear = & operation
To clear 8 bits = 0XFF

x = (x & (~(0XFF<<0) | (0XFF<<24))) | ((x & (0XFF<<0))<<(24-0))
| ((x & (0XFF<<24))>>(24-0));

Swap 1st & 4th byte

```
x = (x & (~(0XFF<<0) | (0XFF<<16))) | ((x & (0XFF<<0))<<(16-0));
```

Swap 1st & 3rd byte

```
(x & (0X FF<<16) >> (16-0));
```

```
x = (x & (~(0XFF<<8) | (0XFF<<24))) | (x & (0XFF<<8))<<(24-8)
```

Swap 2nd & 4th byte

```
(x & (0XFF<<24) >> (24-8));
```

⇒ char x = 0X80;

x = x >> 4;

Printf ("%x\n", x);

signed type

Msb 1 = Add 1's on Msb & Del LSb

Msb 0 = Add 0's on Msb & Del LSb

1000 0000

4 times right shift

Msb 1 ∴ Add 1's on Msb

1111 1000 0000

Add 1's as it is signed type — Del

F 8

⇒ char x = 0X7F;

x = x >> 4;

Printf ("%x\n", x);

0111 1111

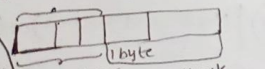
Msb = 0, Add zeroes

0000 0111 — Add zeroes — Del

07

⇒ char x;

scanf ("%d", x);



It tries to print which is not belonging to the current process

⇒ Control statements :-

⇒ If we want to change the flow of execution of the program we have to use control statements.

⇒ In General, CPU will execute the instructions sequentially one after the other.

⇒ changing the flow of the execution is ntg but skip some 'c' stmts (or) repeat the same no. of lines over a period of time (or) you may jump the execution from one location to another location. we can use control statements.

Control statements :-

① selective statements (if, else), (nested if-else, else-if ladder), switch case.

② Ittesative (or) looping stmts (while loop, do-while, for loop).

③ Jump statements. (goto, break, continue, return (fun^{c} stmts))

program logic { operators + control stmts

⇒ Approximately ⑪ keywords will comes under control statements.

if
Else
while
do
for
continue
break
goto
switch
case
default

① Selective statements :-

If-else stmts will comes under the selective stmts.

Syntax :-

```
if (condition)  ------> Relational operators (R + bitwise & logical)
{
   ≡   } If block (True stmts)
}
else
{
   ≡   } else block (false stmts)
}
```

⇒ If group of 'c' stmts written under if condition within pair of curly braces i.e., known as if block.

⇒ Group of 'c' stmts which are written under else stmts within pair of curly braces i.e., known as else block.

⇒ This if-else stmts execution depends upon the condition. If the condition is true it will executes the if block & skips the else block. If the condition is false it will skip the if block & executes the else block.

⇒ we can't write else block separately without if.
- else must be followed by if.
- But we can write only if block with condition by skipping else stmt.

⇒ If single stmt has to be executed under if or else, no need to use pair of curly braces.

⇒ If-else stmts are also known as decision making statements.

✱ write a program to verify the given no. is even (or) odd.

⇒
```
main()
{
 unsigned int x;
 printf("enter the i/p\n");
 scanf("%d", &x);
 if (x%2==0)
    PF("x is even");
 else
    PF("x is odd");
}
```

```
if (x%2)
  PF("odd");
else
  PF("even");
```

✱ write a c program to verify the given value lies b/w 9 to 99.

⇒
```
main()
{
 int x;
 PF("enter the i/p\n");
 SF("%d", &x);
 if((x>=9)&&(x<=99))
 PF("x lies b/w 9 to 99");
 else
  PF("x doesn't lies b/w 9 to 99");
}
```

✱ write a prgm to verify the given no is divisible by 3 & 5.

⇒
```
int main()
{ int x;
  PF("enter the i/p\n");
  SF("%d", &x);
 if ((x%5==0)&&(x%3==0))
 PF("x is not divisible by 3&5");
 else
  PF("x is not divisible by 3&5");
}
```

✱ write a prgm to verify the given character is upper case (or) lower case.

⇒
```
int main()
{ char x;
  PF("enter the i/p\n");
  SF("%c", &x);
 if((x>=65)&&(x<=90))
  PF("It is upper case");
else if((x>=97)&&(x<=122))
  PF("It is lower case");
else
  PF("Not a character")
}
```

(*) write a 'c' prgm to verify the given character is vowel or not.

⇒
```
main()
{
 char x;
 PF("Enter the i|p\n");
 SF("%d", &x);
 if (x=='A' || x=='E' || x=='I' || x=='O' || x=='U')
 PF("It is upper case vowel\n");
 else if (x=='a' || x=='e' || x=='i' || x=='o' || x=='u')
 PF("It is lower case vowel\n");
 Else
  PF("Not a vowel");
```

(*) write a 'c' prgm to verify the given no. is divisible by 5 or not.

⇒
```
int main()
{
 int x;
 PF("Enter the i|p\n");
 SF("%d", &x);
 if (x%5==0)
 printf("It is divisible by 5");
 else
  PF("It is not divisible by 5");
}
```

(*) write a 'c' prgm to verify the biggest among '2' numbers.

⇒
```
int main()
{
 int x=5, y=2;
 if (x>y)
  printf("x is biggest");
 else
 printf("y is biggest");
}
```

12/10/22

⇒ when we are using multiple conditions, we have to use nested if-else statements.

Using only if stmts

```
main()
{
 int a=10;
 int b=20;
 int c=30;
if((a>b)&&(a<c))
 PF("%d\n",a);
if ((b>c)&&(b>a))
 PF("%d\n",b);
if ((c>a)&&(c>b))
 PF("%d\n",c);
}
```

⇒ In all possible cases, it will verify 3 conditions. Eventhough the 1st condition is ~~verified~~ satisfied, it will also verify remaining 2 conditions.

⇒ This code will reduce (or) degrade the performance of CPU, cuz if the first condition is satisfied, it will also verify remaining 2 conditions.

By using Nested if-else

```
if (a>b) ✓
 {
  if (a>c)
  PF("%d\n",a);
  else
   PF("%d\n",c);
 }
else
 {
  if (b>c) ✓
  PF("%d\n",b);
  else
   PF("%d\n",c);
 }
```

⇒ In all possible cases it will verify only 2 conditions.

⇒ when we use Nested if-else stmts we must follow indentation.

⇒ For multiple nested if-else stmts we must use pair of curly braces.

⇒ If we follow the indentation, understanding the flow of execution is easy & also debugging is easy.

⇒ The length of the code will increase (i.e., No. of lines increases) cuz the pair of curly baces are used more, the code will shift towards right hand side. To over-come this we can write program as follows:-

```
if (a>b) {
    if (a>c)
      PF
    Else
}
```

⇒ In Else-if ladder, no need to use pair of curly braces.

⇒ we can reduce/minimize the code shifting towards right hand side by using Else-if ladder.

(*) write a 'c' pgm to print

80 - 100 'A'
70 - 79 - B
60 - 69 C
40 - 59 D
0 - 39 F

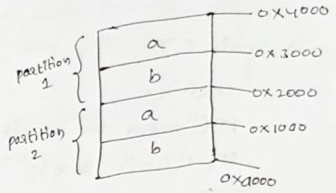
Using Nested if-else

```
main()
{
 int x;
 PF("enter the i/p\n");
 SF("%d", &x);
 if ((x<=100) && (x>=0))
  {
   if (x>=80)
     PF("A");
   else
     { if (x>=70)
       PF("B");
       else
         { if (x>=60)
           PF("C");
           else if (x>=40)
             PF("D");
           else
             PF("F");
         }
     }
  else
    PF("Invalid i/p");
}
```

Using Else-if ladder

```
if((x>=0)&&(x<=100))
  {
   if (x>=80)
    PF("A");
   Else if(x>=70)
    PF("B");
   Else if(x>=60)
    PF("C");
   else if (x>=40)
     PF("D");
   Else
     PF("F");
  }
Else
   PF("Invalid i/p");
```

⇒ write a 'c' pgm to find the partition.




```
main()
{
 unsigned int x;
 printf("enter the address\n");
 scanf("%d", &x);
if ((x>=0x0000)&&(x<=0x4000-1)
                    (0x3FFF)
 {
  if (x>=0x3000) .  ⟶ (0x3000 to 0x3FFF)
  PF("part:1 - a block");
  Else if (x>=0x2000)  — (0x2000 to 0x2FFF)
   PF("part:1 - b block");
  Else if (x>=0x1000)  — (0x1000 to 0x1FFF)
   PF("part:2 - a block");
  Else if (x>=0x0000)
    PF("part:2 - b block");
 }
 Else
   printf("Invalid address");
}
```

(↓ considering for Top to Bottom address)

(or)

```
if ((x>=0x0000)&&(x<=0x3FFF))
  {
   if (x<1000) (or) x<=(0xDFFF)
    PF("part:2 - b block");
   Else if (x<0x2000) (or)(x<=0x1FFF)
    PF("part:2 - a block");
   Else if (x<0x3000) or (x<=0x2FFF)
    PF("part:1 - b block");
   Else
    PF("part:1 - a block");
  }
 Else
   PF("Invalid address");
}
```

(↑ considering for Bottom to Top address)

(*) write a program to print upper case → lower case
lower case → Upper case

⇒
```
main()
{
  char x;
  PF("enter the character\n");
  SF("%c", &x);
  if (((x>=65)&&(x<=90)) || ((x>=97)&&(x<=122)))
  {
    if ((x>=65)&&(x<=90))
    {
      x = x+32;
      PF("%c", x);
    }
    else if ((x>=97)&&(x<=122))
      x = x-32;
      PF("%c", x);
  }
  else
    PF("%t is a special character");
}
```

(or)

```
if ( ---- )
  PF("%c", x+32);
else if ( ---- )
  PF("%c", x-32);
```

(*) (*) ⇒ Iterative (or) looping statements :-

Loops :- loops are also known as iterative stmts.

loops
① while
② do while
③ for

⇒ Display 1 to 10 numbers

```
main()
{
  int x=1;
  PF("%d\n", x);  → 1
  x++;
  PF("%d\n", x);  → 2
  x++;
  PF("%d\n", x);  → 3
  x++;
  .
  .
  .
```

10 times operation
20 lines

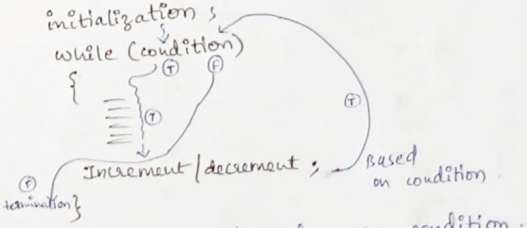
⇒ To reduce this Use loops.

⇒ If we want to repeat or perform a particular task (or) operation continuously over a period of time we have to use loops.
But when we write loops we must take care of
↳ initialization
↳ condition for termination of the loop
↳ Increment / decrement operation.

* advantages :-
① Easy to debug
② Reduce code lines
③ executable file size gets reduced.
④ Modification time & debugging time is less.

① while loop

Syntax :-



⇒ This increment / decrement operation depends upon condition.

* Flow of Execution :-

First it will execute initialization, then verifies the condition, if the condition is true it will execute the loop body, again jumps to the cond". so, if the is true it will execute the loop body, this process continues until the cond" becomes false.

⇒ If the condition is false, before executing the loop the prgm gets terminated.

1 iteration — 1 loop
2 iterations — 2 loops
3 iterations — 3 loops & ----

(*) write a pgm to ~~Verify~~ print 1 to 10 numbers.

⇒ main()
{
int x=1; → verified executes ① time → initialization
while (x<=10) → condition ⇒ verified ⑪ times
{
printf("%d\n", x); → verified ⑩ times
x++; → ⇒ loop body
}
→ increment operation ⇒ verified ⑩ times.
}

x=1	x<=10	printf	x++
✓	1<=10 (T)	1	2
	2<=10 (T)	2	3
	3<=10 (T)	3	4
	4<=10 (T)	4	5
	5<=10 (T)	5	6
	6<=10 (T)	6	7
	⋮	⋮	⋮
	10<=10 (T)	10	11
	11<=10 (F)	–	–
	11 times Verification is done	10 times	10 times

⇒ main()
{
char x=0;
while (x>0)
{
x++;
}
printf("%d\n", x);
}
Output :-
0

⇒ main()
{
char x=1;
while (x>0)
{
x++;
}
PF("%d", x);
}

As it is signed char
1>0 (T)
2>0 (T)
3>0 (T)
⋮
127>0 (T)
128 ⇒ 2's comp −128>0 (F)
∴ it prints 128 in 2's complement
x = −128

⇒ unsigned char x=1;
PF("%d", x);
1>0 (T)
2>0 (T)
⋮
255>0 (T)
256>0 (F)
∴ x=0

⇒ main()
{
int x=9;
while (x--)
{
printf("%d\n", x);
}
}
9 8 7 6 . . . 0
8, 7, 6, ... 0.
output

⇒ main()
{
int x=9;
while (x!=0)
{
PF("%d\n", x);
x=x-2;
}
PF("%d\n", x);
}

x=9 ✓	x!=0	printf	x=x-2	x
	9!=0 (T)	9	x=9-2	7
	7!=0 (T)	7	=7-2	5
	5!=0 (T)	5	=5-2	3
	3!=0 (T)	3	=3-2	1
	1!=0 (T)	1	=1-2	−1
	−1!=0 (F)	−1	=−1−2	−3

Infinite loop

⇒ while (x>=0) ⇒ 1>=0 (T)
1−2=−1
−1>=0 (F)
∴ x=−1
o/p
x=−1

⇒ main()
{
int x=9;
while (x--);
{
PF("%d\n", x);
}
}
9 8 7 6 . . . 0 −1
8 but not substitutes 7 bit " "
o/p
x=−1

It acts as
while (x--)
{
;
}
{
PF("%d\n", x);
}

⇒ main()
{
int x=10;
while (x>0)
printf("%d\n",x);
x--;
}

o/p

10, 10, 10, - - - infinite

⇒ cuz if we don't use pair of curly braces it will print 10 only & does not perform decrement operation (treating printf fun^c as the loop).

⇒ main()
{
int x=1, y=1;
while (y<=5, x<=10)
{
PF("%d-%d\n", x, y);
x++;
y++;
}
}

o/p

(x)
1-1
2-2
3-3
4-4
5-5

(✓)
1-1
2-2
3-3
4-4
5-5
6-6
7-7
8-8
9-9
10-10

⇒ main()
{
int x=1;
while (x++, x<=10)
{
printf("%d\n", x);
}
}

2<=10	2
3<=10	3
4<=10	4
:	:
10<=10	10

o/p = 2 3 4 5 - - - 10.

⇒ cuz comma operator considers right most part

⇒ It will verify y<=5 & also x<=10 acts as cond^n for terminating the loop.

⇒ main()
{
int x=1;
while (x++, x<=10);
{
PF("%d\n", x);
}
}

⇒ It acts as
while (x++, x<=10)
{ ; }
{ PF(-) }

o/p

1
x++ 2, 2<=10 PF 2
3, 3<=10 3
:
10 10<=10 10
11 11<=10 x

Ⓔ

⇒ main()
{
int x=1;
while (x<=10);
{
x++;
PF("%d\n", x);
}
}

o/p

1<=0
1<=0
1<=0

⇒ Improper initialization (or) cond^n, skipping of increment (or) decrement operation may leads to infinite loops.

How do you terminate a prgm which is going into infinite loop

⇒ use ctrl + C (terminate the program) it will generate a signal (SIGINT)

⇒ use ctrl + Z (stops the process execution) it will generate a signal (SIGSTOP).

13|10|22

Ⓐ Write a program to print range of values for a given i/p.

⇒ main()
{
int min, max;
PF("enter the range min:max");
SF("%d-%d", &min, &max);
while (min<=max)
{
PF("%d\n", min);
min++;
}

I/p = 0-10

o/p =
0
1
2
3
4
5
6
7
8
9
10

⇒ If the input is 10-0
use swapping

int min, max, temp;
PF("enter range");
SF("%d-%d", &min, &max);
if (min>max)
temp=min, min=max, max=temp;
while (min<=max)
{ =

(*) write a prgm to print only multiples of '5' in the given range.

⇒
```
main()
{
  int min, max ;
  PF ("enter the i/p range min:max");
  SF ("%d:%d", &min, &max);
  while (min <= max)          ---> verifies (101) times
  {
    if (min % 5 == 0)         ---> verifies (100) times
      printf ("%d\n", min);   ---> 20 times
    min++;                    ---> 100 times
  }
```

I/p
0 - 100

So, to reduce the no. of iterations

```
if (min % 5 != 0)
  min = min + 5 - (min % 5);
while (min <= max)
  {
    PF ("%d\n", min);
    min = min + 5;
  }
```

I/p
4 - 10

4 % 5 != 0 (T)

min = 4 + 5 - (4 % 5)
= 4 + 5 - 4
= 4 + 1
min = 5

(*)

I/p
5 to 100
5 % 5 != 0 (F)
It will skip min stmt
& enters to while loop

I/p
7 to 100
7 % 5 != 0 (T)

min = min + 5 - (min % 5)
= 7 + 5 - (7 % 5)
= 7 + 5 - 2
= 7 + 3
min = 10

verifies from 10 to 100.

(*) write a prgm to count the no. of digits in a given number.

⇒
```
main()
{ int x, count = 0;
  PF ("Enter the i/p\n");
  SF ("%d", &x);
  if (x != 0)
  {
    while (x > 0)        ---> It is only for +ve no. verification.
    { x = x / 10;        ---> (x != 0) ↳ For both +ve & -ve no. verification.
      count ++;
    }
    printf ("%d\n", count);
  }
Else
  count ++;
}
```

I/p - 123
O/p - 3 (digits)

(*) write a prgm for to print a given range of value no & no. of digits

⇒
```
main()
{
  int min, max, num, count = 0;
  PF ("Enter the i/p\n");
  SF ("%d-%d", &min, &max);
  while (min <= max)
  {
    printf ("%d - ", min);
    if (min != 0)
    {
      num = min;
      while (num != 0)
      {
        num = num / 10;
        count ++;
      }
    }
    Else
      count ++;
    printf ("%d\n", count);
    count = 0;
    min ++;
  }
```

0 != 0 (F) Exit

1 != 0	8 != 0
1/10 = 0	8/10 = 1
c = 2	c = 1

8 <= 12 (T)
8 - 1
8 != 0 (T)
num = 8
8 != 0 (T)
8 = 8/10 = 1
c++ = 1

I/p = 8-12
O/p = 8 - 1
9 - 1
10 - 2
11 - 2
12 - 2

Extract & add the given no.

⇒
```
main()
{
  int num, sum=0, rem, temp;
  PF("Enter the i/p\n");
  SF ("%d", &num);
    temp = num;
    while (temp != 0)
    {
      rem = temp % 10;
      temp = temp / 10;
      sum = sum + rem;
    }
  printf("%d\n", sum);
}
```

I/p = 123

O/p

123 % 10 = 3
123 / 10 = 12
12 % 10 = 2
12 / 10 = 1
1 % 10 = 1
1 / 10 = 0

1+2+3 = 6
3+2+1 = 6

Extract and add for the range of numbers.

⇒
```
main()
{
  int min, max, sum=0, rem, temp;
  PF("Enter the i/p\n");
  SF ("%d - %d", &min, &max);
    while (min <= max)
    {
      sum = 0;
      temp = min;
      while (temp > 0)
      {
        rem = temp % 10;
        temp = temp / 10;
        sum = sum + rem;
      }
      printf("%d\n", sum);
      min++;
    }
}
```

Input :- 121 : 125

min = 121
max = 125

121 <= 125 (T) outer loop
sum = 0

121 > 0 (T)
rem = 121 % 10 = 1
temp = 121 / 10 = 12
sum = 0 + 1 = 1

12 > 0 (T)
rem = 12 % 10 = 2
temp = 12 / 10 = 1
sum = 1 + 2 = 3

1 > 0 (T)
rem = 1 % 10 = 1
temp = 1 / 10 = 0
sum = 3 + 1 = 4

0 > 0 (F)
Sum = 4
outer loop

Inner loop

min++;
min = 122;
122 <= 125 (T)
sum = 0

122 > 0 (T)
rem = 122 % 10 = 2
temp = 122 / 10 = 12
sum = 0 + 2 = 2

12 > 0 (T)
rem = 12 % 10 = 2
temp = 12 / 10 = 1
sum = 2 + 2 = 4

1 > 0 (T)
rem = 1 % 10 = 1
temp = 1 / 10 = 0
sum = 4 + 1 = 5

0 > 0 (F)
sum = 5

Inner loop

min++;
min = 122 + 1 = 123;
123 <= 125 (T)
sum = 0

123 > 0 (T)
rem = 123 % 10 = 3
temp = 123 / 10 = 12
Sum = 0 + 3 = 4

⇒ Write a program to find whether the given no is palindrome or not.

⇒
```
main()
{
  int num, sum=0, rem, temp;
  PF("enter the i/p");
  SF ("%d", &num);
    temp = num;
  while (temp > 0)
  {
    rem = temp % 10;
    temp = temp / 10;
    Sum = sum * 10 + rem;
  }
    If (sum == num)
  printf("%d\n", sum);
```

I/p = 123

123 % 10 = 3
123 / 10 = 12
sum = 0 * 10 + 3 = 3
12 % 10 = 2
12 / 10 = 1
sum = 3 * 10 + 2 = 32
1 % 10 = 1
1 / 10 = 0
sum = 32 * 10 + 1
= 321

not a palindrome

(*) write a program to display only the palindrome in a given range of values.

⇒
```
#include <stdio.h>
int main()
{
    int min, max, sum=0, rem, temp;
    PF("enter the range min:max\n");
    SF("%d:%d", &min, &max);
    while (min<=max)
    {
        sum=0;
        temp= min;
        while (temp>0)
        {
            rem= temp%10;
            temp = temp/10;
            sum= sum*10+rem;
        }
        if(sum == min)
        printf("%d\n", sum);
        min++;
    }
}
```

I/P :- 20:50
O/P = 22
33
44

(*) write a pgm to print given decimal into binary without using bitwise operators.

⇒
```
main()
{
int num, sum=0, rem, temp, i=1;        → powers of 10.
PF("I/p");
SF("%d", &num);
  temp = num;
while (temp!=0)
{
   rem = temp%2;
   temp= temp/2;
   sum= sum+i*rem;
    i=i*10;
}
printf("%d\n", sum);
}
```

2|10
2|5 - 0 → $*10^0$
2|2 - 1 → $*10^1$
1 - 0 → $*10^2$
↓
$*10^3$

⇒ Binary to decimal

⇒
```
main()
{
int num, rem, sum=0, rem, temp, i=1;
PF("I/p");
SF("%d", &sum);
  temp =num;
while (temp!=0)
{
  rem= temp%10;
  temp= temp/10;
  sum = sum*i*rem;
   i= i*2;
}
PF("%d\n", sum);
}
```

1101 ⇒ 13 = (D)

(*) Break statement :-

⇒ It is a jump statement which will terminate that particular loop there itself.

⇒ It must be used along with proper condition.

Eg:-
```
while (x)
{ if (x%5==0)
    break;
  printf("%d\n", x);
  x++;
}
```
comes out from the loop.

✱ Binary to decimal using break statement

⇒
```
int num, rem, sum=0, temp, i=1;
{
  PF("i/p\n");
  SF("%d", &num);
   temp = num;
  while (temp!=0)
  {
     rem = temp%10;
      if (rem>1)
      {
        sum=0;
        PF("Invalid i/p\n");
        break;
      }
      temp = temp/10;
      sum = sum + i*rem;
       i = i*2;
  }
  printf("%d\n", sum);
}
```

5 0 1
↓
break

✱ Continue statement :-

⇒ Continue statement is also a jump statement.

⇒ It must be used along with proper condition with increment or decrement operation.

⇒ This stmt is used in loops to skip one iteration.

⇒ When CPU executes this continue stmt, the control is transfered from current stmt to same loop conditional part. But in For loops, it will jump to the increment or decrement statement.

✱
```
while (condition)
{
  if (cond^n)
  continue;
}
```

⇒
```
For (initialization; condition; increment/decrement)
{
  if (condition)
  continue;
}
```

Jumps to inc/dec but not enters into condition.

⇒ In while loop if we want inc/dec we need to use inc/dec before continue stmt.

```
while (condition)
{
  if (cond^n)
  {
    inc/dec;
    continue;
  }
}
```

→ Inc/dec can be done here.

NOTE:-

⇒ The statements written below continue are not executed.

✱ In a given range of values, skip multiples of 5.

⇒ I/p = 2:17
O/p = 2 3 4 6 7 8 9 11 12 13 14 16 17

⇒
```
main()
{
  int min, max, temp;
  PF("Enter the range\n");
  SF("%d:%d", &min, &max);
  while (min<=max)
  {
    if (min%5==0)
    {
      min++;
      continue;
    }
    PF("%d\n", min);
    min++;
  }
```

⊙ Add any printf stmt to verify (it will not print after continue stmt).

(*) For a given binary range of values print decimal no's.

Assignment

use break & continue

I/p	o/p
1:1000	1-1 10-2 11-3 100-4 101-5 110-6 111-7 1000-8.

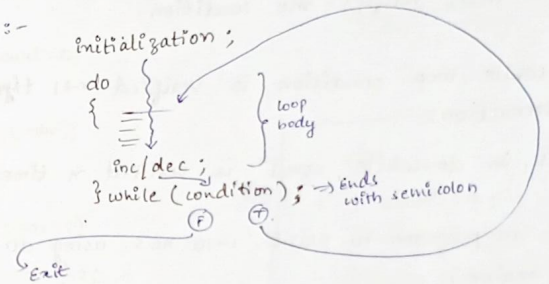
```
⇒ main()
{
  int min, max, bin, dec=0, i=1;
  PF("enter the i/p min:max\n");
  SF("%d:%d", &min, &max);
  while (min<=max)
  {
    dec=0;
    i=1;
    bin=min;
    while (bin>0)
    {
      rem=bin%10;
      if (rem>1)
      {
        break;
      }
      bin=bin/10;
      dec=dec+rem*i;
      i=i*2;
    }
    if (rem>1)
    {
      min++;
      continue;
    }
    Printf("%d-%d\n", min, dec);
    min++;
}
```

1-1
(2,3,4,5---9) skip
10-2
11-3
(12-13,14,15---99) skip
100-4
101-5
(102-103---109) skip
110-6
111-7
(112---999) skip
1000-8

14/10/22

② do-while

Syntax :-



(*) Flow of execution :-

First, it will executes the initialization & directly executes the loop body including increment (or) decrement operation, then verifies the condition. If it is true the process will continue until the cond^n becomes false.

(*) Differences between while & do while :-

① ⇒ In while loop, while followed by the condition will not end with semi colon. If we use semicolon it goes to infinite loop.

```
x=10;
while (x--);
printf("%d\n",x);
```
→ o/p (x=-1) not goes to infinite loop (rare cases)

⇒ Whereas in do while, we must use semi colon after while cond^n. If we don't use the semi colon after condition we will get compilation Error.

② ⇒ In while loop, if the cond^n is false for the first time, the loop gets terminated there itself.

⇒ In do while, if the cond^n is false for the first time, the loop body gets executed one time, cuz all the stmts are written above the condition.

③ ⇒ In while loop, after initialization it verifies the cond^n & then executes the body.

⇒ whereas, in do-while, After initialisation it executes the loop body & then verifies the condition.

④ ⇒ In while loop condition is verified n+1 times for n itterations

⇒ whereas, in do-while, cond^n is verified n times for n itterations.

(*) write a program to print 1–10 no's using do-while.

⇒
```
main()
{
  int x=1;                    ← 1 time
  do
  {
    printf("%d\n", x);        } → 10 times
    x++;
  } while (x<=10);            → 10 times
}
```

x=1	printf	x++	x<=10
✓	1	2	2<=10 (T)
✓	2	3	3<=10 (T)
	3	4	4<=10 (T)
	⋮	⋮	⋮
	9	10	10<=10 (T)
	10	11	11<=10 (F)

⇒ condition is verified 10 times

(*) write a program to count the no. of digits for a given no. using do while.

⇒ (using while)

```
main()
{
  int num, count=0;
  PF("Enter");
  SF("%d",&num);
  if (num!=0)
  {
    while (num!=0)
    {
      num = num/10;
      count++;
    }
    printf("%d\n", c);
  }
  else
  {
    count++;
    PF("%d\n", c);
  }
```

Using do-while

```
do
{
  count++;
  num = num/10;
} while (num!=0);
printf("%d\n", c);
```

⇒ do-while is used without if condition

(*) Print the sum until the user enters '0' using while, do while & break stmts.

⇒ using do-while

```
main()
{
  int num, sum=0
  do
  { scanf("%d", &num)
    sum = sum+num;
  } while (n!=0);
  printf("%d\n", sum);
}
```

using while → to avoid garbage value

```
int n=1, sum=0;
{SF("%d", &n);
while (n!=0)
  {
    scanf("%d", &n);
    sum = sum+n;
  }
printf("%d\n", sum);
}
```

```
int n, sum=0
scanf("%d", &n);
while (n!=0)
{
 sum=sum+n;
 SF("%d", &n);
}
```

Using break stmt

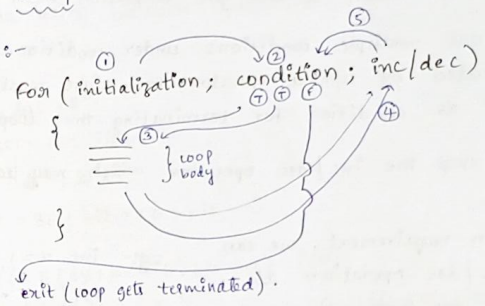
```
int n, sum=0;
while (1) ——————→ It will become true always
{
  scanf ("%d", &n);
  if (n==0)
    break;
  sum = sum+n;
}
```

* write a pgm to find whether the given no is palindrome or not using nested do while for a given range of values

⇒
```
main()
{ int num, min, sum=0, rem, max;
      PF(-);
  do  SF(-);
  {
    num = min;
    sum = 0
    do
    { rem = num % 10;
      num = num / 10;
      sum = sum + rem;
    } while (num>0);
    PF("%d\n", sum);
    min++;
  } while (min <= max);
```

⇒ ③ For loop :-

Syntax :-



* Flow of Execution :-

⇒ The flow of execution is same as while loop. First, it will executes the initialization & verifies the cond^n, if it is true then it enters to the loop body & it perform increment and decrement operation until the cond^n becomes false.

⇒ Under initialization of for loop we should not declare (or) define a variable which is not supported by c89 standard but it is supported in c99 standard.

initialization in → c89 standard → not supported
→ c99 standard → int x=0; supported.

⇒
```
main()
{
  int x;
  for (x=1; x<=10; x++)
    printf ("%d\n", x)
}
```

```
int x=1;
for (    ; x<=10; x++)
  printf ("%d\n", x);
```

If we dont write anything under initialization still it is valid.

⇒
```
int x=1;
for (    ;    ; x++)
```

⇒ we can skip the conditional part that indicates always true statement which leads to infinite loop.

⇒ while ( )

↳ If we skip it will give compilation error.

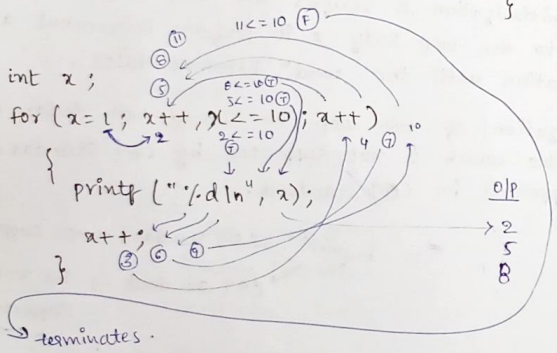
⇒ we can use multiple conditions under conditional part which are separated by comma operator, the right most condition is considered as condition for terminating the loop.

⇒ we can skip the inc/dec operator which may leads to infinite loop.

⇒ Based on requirement, we can write inc/dec operation as part of loop body.

eg:-
```
int x=1;
for(    ; x<=10;    )
{
  PF("%d\n", x);
  x++;
}
```

⇒
```
int x;
for(x=1; x++, x<=10; x++)
{
  printf("%d\n", x);
  x++;
}
```



o/p
2
5
8

Ⓐ For a given range of values convert decimal to Binary.

⇒
```
int rem, min, max, bin, dec;
PF("i/p range");
SF("%d:%d", &min, &max);
for(min; min<=max; min++)
{
  for(dec=min, bin=0, i=1; dec>0; dec=dec/2)
  {
    rem=dec%2;
    bin=bin+i*rem;
    i=i*10;
  }
  printf("%d-%d\n", min, bin);
}
```

Ⓐ write a program to print the binary values stored for a given value using bit wise operators

⇒
```
main()
{
  int x, bit;
  PF("enter the i/p\n");
  SF("%d", &x);
  for(bit=31; bit>=0; bit--)
  {
    if((x&(0x1<<bit))==0)
      PF("0");
    else
      printf("1");
  }
  printf("\n");
}
```

(or)

```
for(bit=31; bit>=0; bit--)
  printf("%d\n", ((x>>bit)&(0x1)));
```

Ⓐ Count no. of zero's and one's in the binary number.

⇒ x = 0x11223344

0001 0001 0010 0010 0011 0011 0100 0100 → 22 = 0's
10 = 1's

```
z=0, o=0;
for(bit=31; bit>=0; bit--)
{
  if((x&(0x1<<bit))==0)
    z++;
  else
    o++;
  printf("%d\n", count);
}
```

⊛ Write a program to swap all the bits from MSB to LSB.

0001 0001 0010 0010 0011 0011 0100 0100

⇒
```
main()
{
  int i, j, x;
  PF(" Enter the i/p\n");
  SF("%d", &x);
  for (i=0, j=31; i<j; i++, j--)
    x = (x & (~(0x1<<i) | (0x1<<j))) | ((x & (0x1<<i)) << (j-i)) |
        (x & (0x1<<j)) >> (j-i));
  printf("%d\n", x);
}
```

⊛ Count how many 101's are there in the given no.

0101 0101 0010 1010 0011 0011 0100 0100

6 ⇒ 101's are present.

⇒
```
main()
{
  int i, x, c;
  PF(" o/p");
  SF("%d", &x);   (32 bit | 8=8 bit | 16 for 16 bit) → no. of digits to be verified.
  for (i=0; i<=(32-3); i++)
  {
    if ((x & (0x7)<<i)) == (0x5))   → 0x7: mask value based on combination of bits
      c++;
  }
```

For 8 bit

10101010 — 3 times

```
for (i=0; i<=(8-3); i++)
{
  if ((x & (0x7 <<i))==
      (0x5))
    c++;
```

```
10101010
     111         i=0
00000010         x=5
                 → compared
 010 == 101
    (F)
```

Using right shift operator

```
for (i=32; i>=3; i--)
  if ((x>>(i-3)) && (0x7)) ==(0x5)
    c++;
```

⇒ Extract & print odd & even place values

```
x = 10 10 10 10;
y = 0;
for (i=0, j=0; i<8; i=i+2, j++)
{
  if ((x & (0x1<<i)) == 0)
    y = y & (~(0x1<<j));
  Else
    y = y | (0x1<<j);
}
```
} To Read & separate even positions

o/p: 1111 0000

```
for (i=1; i<8; i=i+2, j++)
{
  if ((x & (0x1<<i)) == 0)
    y = (y & (~(0x1<<j)));
  Else
    y = (y | (0x1<<j));
}
```
} To read & separate odd positions

15/10/22

* Patterns :-

⇒ To print 5 stars ⇒ `for (i=0; i<5; i++) printf("*");`

o/p: * * * * *

→ Print

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

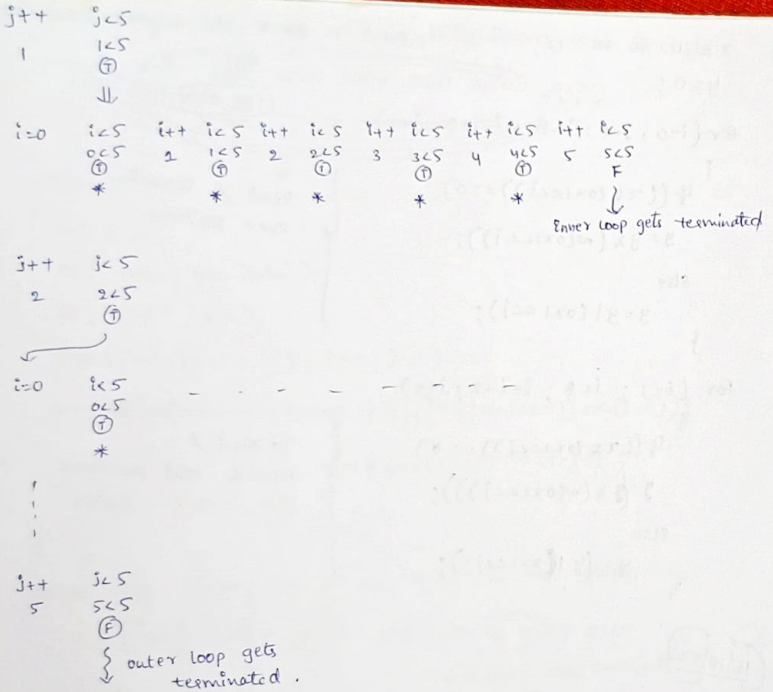
⇒ Nested for loop

```
{
for (j=0; j<5; j++)
{
  for (i=0; i<5; i++)
    Print("*");
  }
  Print("\n");
}
```

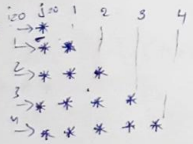
Explanation

Outer loop: j=0, j<5 → 0<5 (T) ↓ enters inner loop

i=0	i<5	i++	i<5	i++	i<5	i++	i<5	i++	i<5	i++	i<5
	0<5	1	1<5	2	2<5	3	3<5	4	4<5	5	5<5
	(T)		(T)		(T)		(T)		(T)		(F)
	It prints *		*		*		*		*		↓ Loop gets terminated

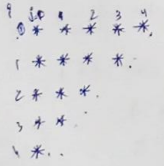


⇒ paint



```
for (j=0; j<5; j++)
 {
  For (i=0; i<=j; i++)
    Printf ("*");
    Printf ("\n");
  }
```

⇒ paint

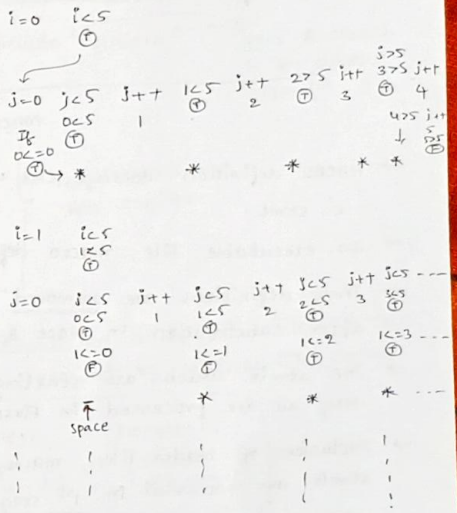


```
for (j=5; j>0; j--)
 {
  For (i=0; i<j; i++)
    Printf ("*");
    Printf ("\n");
  }
```

⇒ paint

```
for (i=0; i<5; i++)
 {
  for (j=0; j<5; j++)
    if (i<=j)
      Printf ("*");
    Else
      Printf ("  ");
  Printf ("\n");
 }
```

Explanation.



17/10/22

* MACRO's

⇒ Macros are defined by pre-processor directive on the top of the program after header files inclusion.

⇒ MACRO name should be mentioned always in CAPITAL letters. Pre-defined MACRO definitions are present in header files. When we are using the pre-defined macros in our prgm we must include that in the corresponding files.

⊛ NOTE :-

⇒ Even Hardware register's address are defined by the MACRO's in header files.

```
# define UART0_DATA (volatile)*((unsigned char*)(0x8000F000))
```

(or)

```
((volatile unsigned char*)0x8000F000)
```

Syntax :-

```
# define <MACRO Name>  <expression>
                       constants
                       c-code.
```

MACRO definition.

⇒ MACRO definition doesn't end with semi colon, coz it is not a 'c' stmt.

⇒ In executable file, macro definitions doesn't occupy any memory.

⇒ These definitions are removed in first stage of compilation after substitution in place of macro names.

⇒ The stmts which are starting with pre-processor directive (#) they all are processed in first stage of compilation.

⇒ Inclusion of header files, macro definitions & conditional compilation stmts are processed in 1st stage of compilation. Apart from this even comments are removed in 1st stage of compilation.

```
# if def
# end if
# include <stdio.h>
# define MACRO 5.
```

```
//  ⇒ single line comments
/*
 =
 =   ⇒ multiple line comments.
 =
*/
```

⇒ we can stop the compilation after 1st stage to check the pre-processed output

** $ gcc -E <c prgm file name> >

⇒ This command will stop the compilation process after 1st stage and displays that pre-processed output on the terminal screen itself.

⇒ # include <stdio.h> will be replaced with the content of the header file in 1st stage of compilation.

/usr/include (default path) ← goes & search in default path (< >)

include <stdio.h>

include "stdio.h" → goes & search in the current prgm (" ").

⇒ c-library
⇒ Architecture
  → little Endian/Big Endian
  → Architecture info
⇒ Kernal header

} requirements for compiler.

⇒ To copy the pre-processed output in a separate file we can use redirect operator (>) followed by textual file name

** $ gcc -E sample.c > sample.i
(c prgm file name) (textual file name)

⇒ Here, Now the pre-processed o/p will not displays on the terminal screen, it copies the content & saves in sample.i textual file.

⇒ If the file is already exists it will over write the content. If the file doesn't exist, it will create a new file & copies the content.

⇒ $ ./a.out > <textual file name>.

Now, the overall o/p will not display on the terminal screen but copied & redirected to separate textual file.

⇒
```
# define MAX 5
# define STR "welcome".
main()
{
  printf("%d\n", MAX);   → Max will be replaced by 5
  Printf(STR);           → STR will be replaced by "welcome".
```

```
# define F(x)  (x)*(x)
main()
{
  int x=10;
  y=F(x);
  PF("%d\n",y);
}
```

⇒ O/P

y = (x) * (x)

= 10 * 10

y = 100

⇒

```
# define F(x)  (x)*(x)
main()
{
  int x=10;
  y=F(x+1);
  PF("%d\n",y);
}
```

⇒ O/P

y = x+1 * x+1

(14) (4) (3) (4) = priority values

= 10+1 * 10+1

= 10+10+1

y = 21

⇒ whenever we write the expressions by using MACRO's we must use paranthesis at appropriate locations to avoid False outputs.

eg:- # define F(x) x*x — ✗

define F(x) (x)*(x) — ✓

⊛ Switch Case :-

gcc -E
gcc -o
gcc -s
gcc -v
→ case labels

gcc → executable files

⇒ On a particular data, if we want to perform multiple operations & these operations are to be selected by the user. so, in that case we will use switch.

⇒ In General, to write menu driven programs to implement commands with various flags or options, to select the functionality of the device from the user space.

⇒ Keywords used are switch, case, default & break.

Syntax :-

```
switch (opt)
{
  case label 1:
    ===  } task-1
    break;
  case label 2:
    ===  } task-2
    break;
    .
    .
  default :
    ===
}
```

label 1 → we can use character constant / Integer const / MACRO name

Integer constant

In driver level we write macros as labels

break; → termination of case 1

⇒ Each case will define a particular task.

⇒ Label can be a character const / integer const / MACRO name.

⇒ That is a Fixed constant value.

⇒ Label should be followed by colon (:).

Break importance in switch case :-

⇒ It will stop the execution of the switch case (Termination of switch case).

⇒ we are reading the case label as i/p in the option, this is compared with the each case label, if it finds the matching case label it will executes that particular task. If it doesn't find the matching case label then it will execute default case.

⇒ Each case label should be defined only one time.

⇒ we can't define two case labels with the same constants or names or numbers.

⇒ when we use ASCII characters as case labels, we have to read the i/p in ASCII characters only.

```
char opt;
scanf("%d", &opt);
switch (opt)
{
  case 'A' : printf("Hi");
  (65)       break;

  case 'B' : printf("Hello");
  (66)       break;
}
```

Integer value

⇒ case 1 :

⇒ case '1' :

(49) ASCII character value

⇒ case A :-

compilation error cuz A is not a integer constant / nor character const if we use like this.

⇒ If we want to terminate the switch case (or) loop, we have to use break statement.

⇒ If we want to terminate the whole program, we have to use exit(0) function.

⇒ If we want to skip some 'c' stmts, we have to use continue statement.

Assignment - ①

① Reverse a no.
② sum of digits
③ count no. of digits
④ palindrome or not
⑤ Binary ~~to decimal~~
⑥ Even or odd.

Assignment - ②

① count no. of 0's & 1's.
② 101 repeated ?
③ print the binary values
④ swap all the bits.
⑤ Replace 101 with 111

④
```
#include <stdio.h>
main()
{
  int opt, num, temp, sum=0, rem;
  printf("select the option\n 1. Reverse a no.\n 2. sum of digits\n
        3. count no. of digits\n 4. palindrome or not\n 5. ~~Print~~ Binary
  ~~to decimal~~ numbers\n 6. Even or odd\n");
  scanf("%d", &opt);
  __fpurge(stdin);   ---> It will clears the standard i/p o/p file descriptor (buffer)
  printf("enter the value\n");
  scanf("%d", &num);
  switch (opt)
  {
    case 1:                          // reverse a no.
      temp = num;
      sum = 0;
      while (temp > ~~num~~ 0)
      {
        rem = temp % 10;
        temp = temp / 10;
        sum = sum * 10 + rem;
      }
      printf("%d\n", sum);
      break;

    case 2:
      temp = num;                    // sum of digits
      sum = 0;
```

```
while (temp>0)
{
   rem= temp %10 ;
   temp = temp /10 ;
   sum= sum+rem ;
}
printf ("%d\n", sum);
break;
```

Case 3 : // count no. of digits

```
temp = num ;
count =0;
while (temp>0)
{
  temp = temp/10 ;
  count ++ ;
}
printf ("%d\n", count);
break;
```

case 4 : // palindrome or not

```
temp = num ;
sum = 0 ;
while (temp>0)
{
   rem = temp %10 ;
   temp= temp/10 ;
   sum = sum *10 +rem ;
}
if (sum == num)
  printf (" palindrome --");
Else
  printf (" Not a palindrome --");
  printf ("%d ", sum);
  break;
```

Case 5 : // print Binary no's

```
temp = num ;
sum=0 ;
while (temp>0)
{ rem = temp%2 ;
  temp = temp/2 ;
  sum = sum + rem * i ;
   i = i*10 ;
}
printf ("%d", sum);
break;
```

// Even (or) odd

case 6 :

```
temp = num ;
if (temp%2==0)
   printf ("Even");
else
   printf ("odd");
break;
```

```
default : printf (" Invalid i/p \n");
  }
}
```

Assignment - ②

```
# Include <stdio.h>
main()
{
  int opt, bit, x , i=0, j=0 , z, count =0;
  PF(" select the option\n 1. count no. of 0's & 1's\n 2. 101 repeated?
     3. print binary values \n 4. swap all the bits \n 5. replace 101
     with 111\n");
  SF ("%d", &opt);
  fpurge (stdin);
  Switch (opt)
   {
     case 1 :                          // count no of 0's & 1's
          PF ("Enter the i/p\n");
          SF ("%x", &x);
          for ( bit =0 ; bit<=31; bit++)
            { if ((x&(0x1<<bit))==0)
```

```
        else
            i++;
    }
    PF("zeroes --- %d  one's -- %d", z, i);
    break;

case 2:                                  //101 repeated?
    PF("enter the i/p\n");
    SF("%x", &x);
    for (i=0; i<=32-3; i++)
    {
      if ((x&(0x7<<i)) == (0x5))
          count++;
    }
    PF("%d\n", count);
    break;

case 3:                                  // print binary values
    PF("enter i/p\n");
    SF("%d", &x);
    for (bit=31; bit>=0; bit--)
    {
      if ((x&(0x1<<bit))==0)
          PF("0");
      else
          PF("1");
    }
    printf("\n");
    break;

                                         // swap all the bits.
case 4:
    PF("enter the i/p\n");
    SF("%x", &x);
    for (i=0, j=31; i<j; i++, j--)
    x = (x & (~(0x1<<i) | (0x1<<j))) | ((x&(0x1<<i))<<(j-i))
            (x&(0x1<<j))>>(j-i));
    PF("%x", x);
    break;
```

```
case 5:                                  // replacing 101 with 111
    PF("enter the i/p\n");
    SF("%x", &x);
    for (i=32-3; i>=0; i--)
    x = ((x&(0x7<<i)) == (0x5)) ? x=0x7 : x;
    PF("%x", x);
    break;

default: PF("Invalid i/p\n");
    }
}
```

18|10|22

⇒ In menu driven programs we have to use infinite while loop & define one option to terminate the program whenever user wants.

exit(0); or return 0; (NOT recommended.)

⇒ return stmt just terminates the particular fun^c, it will not terminate the entire process.

⇒ Using return stmt in main() fun^c it will terminates that fun^c so the complete process will terminates.

```
main()
{
  ≡
  ≡
  return 0;
}
```

⇒ main() fun^c with some statements & return 0; will terminates the main fun^c that leads to terminate the process.

NOTE:-

⇒ return stmt will terminate only the particular fun^c

⇒ exit(0) will terminates the complete process.

⇒

```
# define  SET   1
# define  CLR   2
# define  EXIT  0
# define  MASK  0X01
# define  SET_BIT  (X, BIT)        X | (MASK<<BIT)
                     variable
# define  CLR_BIT  (X, BIT)        X & (~(MASK<<BIT))

main()
{
  unsigned int x;
  unsigned int bit;
  int opt;
  while (1)
  {
    PF("0.exit\n1.set a bit\n2.clear a bit\n select the option\n");
    SF("%d", &opt);
    -- fpurge(stdin);
    PF("Enter the value & bit  value:bit (0 to 31)\n");
    SF("%x:%d", &x, &bit);
    if(!(bit<=31)&&(bit>=0))
    {
      PF("Invalid i/p-bit\n");
      continue;
    }
    switch(opt)
    {
      case EXIT : exit(0);
                  replaced by 0
      case SET  : x = SET_BIT(x,bit);
                  replaced by 1     x|(0x1<<bit) ⇒ replaced in 1st stage of compilation
                  printf("%x\n", x);
                  break;

      case CLR  : x = CLR_BIT(x,bit);
                  replaced by 2     replaced with x&(~(0x1<<bit)); in 1st stage of compilation.
                  printf("%x\n", x);
                  break;
      default : PF("Invalid i/p\n");
    }  → switch case termination
  }  → infinite while loop
}  → main().
```

⇒ without using switch case we can write menu driven prgms by using else-if ladder.

```
if(!(bit<=31)&&(bit>=0))
{
  PF("Invalid i/p-bit\n");
  continue;
}
if(opt==EXIT)
{
  exit(0);
}
else if(opt==SET)
{
  x=SET_BIT(x,bit);
  PF("%x\n",x);
}
else if(opt==CLR)
{
  x=CLR_BIT(x,bit);
  PF("%x\n",x);
}
else
  PF("Invalid option\n");
```

⇒ By using Nested switch case we can have so many options in main options. Nested switch case consists of switch case within switch case.

⇒
```
int opt, num, sub_opt;
Unsigned int bit, sub-opt;
while (1)
 {
   printf ("0.exit\n 1. bitwise operation\n 2. Arithmatic operation\n
          select the option\n");
   Scanf ("%d", &opt);
    switch (opt)
    {
      case 0 : exit(0);
      case 1 : PF("select the option to perform bitwise
                 operations\n a.set \n b. clear\n");
               SF ("%c", sub_opt);
               __fpurge (stdin);
               PF("Enter the values: bit\n");
               SF ("%x : %d.", &num, &bit);
      switch (sub_opt)
       {
         case 'a' : num=num | (0x1<<bit); (or) num = SET_BIT(
                                                num, bit);
                    PF ("%x\n",num);
                    break;
         case 'b' : num=num & (~(0x1<<bit));
                    PF("%x\n", num);
                    break;
         default : PF ("Invalid option");
       }
      break;   //***.
      case 2 : PF ("select the option to perform arithmetic
                 operation\n a. add\n b. sub\n");
               SF ("%c", &sub_opt);
               __fpurge (stdin);
               PF ("enter the i/p's x: y\n");
               SF ("%d: %d", &x, &y);
               switch (sub-opt)
               {
                 case 'a' : z=x+y;
                            PF ("%d\n", z);
                            break;
                 Case 'b' : z=x-y;
                            PF("%d\n", z);
                            break;
                 default : PF("Invalid i/p\n");
               }
               break;
               default : PF ("Invalid option\n");
      }
 }
```

⇒ Paint the following using Nested switch case :-

① Bitwise operations
- a. count no. of 0's & 1's
- b. Paint binary values
- c. swap all the bits
- d. Replace 101 → 111

② Mathematical operations
- a. palindrome
- b. sum of digits
- c. Binary ⟷ decimal.

③ swap the content
- a. without using 3rd variable
- b. with using 3rd variable
- c. Using bit wise operators

④ ASCII operations
- a. Upper case ⟷ lower case
- b. character const ⟷ Integer const.

⑤ Fibonacci series.

```
#include <stdio.h>
void main()
{
 unsigned int x,y,z,j,t,opt,sum=0,num,rem,temp;
 Unsigned char suboptions,k;
 int i, bit;
PF("0. exit\n 1. Bitwise operation\n 2. Mathematical operations\n
   3. Swapping the content\n 4. ASCII operations\n 5. Fibonacci
   Series\n select the option : \n");
SF("%d",&opt);
__fpurge(stdin);
 switch(opt)
  {
    case 1: PF("select the options to perform bitwise
            operations\n a. count no. of 0's & 1's \n
            b. print binary values \n c. swap the content\n
            d. replace 101 with 111 \n");
            SF("%c",&suboptions);
            __fpurge(stdin);
            switch(suboptions)
            {
              case 'a': PF("Enter the i/p\n");
                        SF("%x",&x);
                        for(bit=0; bit<=31; bit++)
                        {
                          if((x&(0x1<<bit))==0)
                            z++;
                          else
                            i++;
                        }
                        PF("zeroes---%d ones--%d\n");
                        break;
              case 'b': PF("enter the i/p\n");
                        SF("%d",&x);
                        for(bit=31; bit>=0; bit--)
                        ((x&(0x1<<bit))==0)? PF("0"): PF("1");
                        printf("\n");
                        break;
              case 'c': PF("Enter the i/p\n");
                        SF("%x",&x);
                        for(i=0, j=31; i<j; i++, j--)
                        x=(x&(~(0x1<<j))|(0x1<<i)))|((x&(0x1<<i))<<(j-i))|
                          (x&(0x1<<j))>>(j-i);
                        PF("%x",x);
                        break;
              case 'd': PF("Enter the i/p\n");
                        SF("%d",&x);
                        for(i=31-3; i>0; i--)
                        x=(((x&(0x7<<i))==0x5))? x=0x7 : 0;
                        PF("%x",x);
                        break;
              default: PF("Invalid i/p\n");
            }
            break;
    case 2: PF("select the options to perform mathematical operations
              a. palindrome or not \n b. sum of digits \n c. binary to
              decimal\n");
            SF("%c",&suboptions);
            __fpurge(stdin);
            PF("Enter the i/p\n");
            SF("%d",&num);
            switch(suboptions)
            {
              case 'a': temp=num;
                        sum=0;
```

```
while (temp > 0)
{
    rem = temp % 10;
    temp = temp / 10;
    sum = sum * 10 + rem;
}
if (sum == num)
    PF (" palindrome --- ");
else
    PF (" Not a palindrome --- ");
    PF ("%d \n", sum);
    break;
```

```
case 'b' : temp = num;
           sum = 0;
           while (temp > 0)
           {
             rem = temp % 10;
             temp = temp / 10;
             sum = sum + rem;
           }
           PF ("%d", sum);
           break;
```

```
case 'c' : temp = num;
           sum = 0;
           i = 1;
           while (temp > 0)
           {
             rem = temp % 10;
             temp = temp / 10;
             sum = sum + rem * i;
             i = i * 2;
           }
           PF ("%d \n", sum);
           break;
default : PF (" Invalid i/p \n");
```

```
case 3: PF (" select the options to swap the content opeeation \n
         a. without using 3rd variable \n  b. By using 3rd variable \n
         c. By using Bitwise opeeators \n");
SF ("%c", suboptions);
--fpurge (stdin);
PF (" Enter the i/p \n");
SF ("%d, %d, &x, &y);
switch (sub options)
{
  case 'a' : x = x + y;
             y = x - y;
             x = x - y
             PF ("%d, %d", x, y);
  break;
```

```
case 'b' : temp = x;
           x = y;
           y = temp;
           PF ("%d, %d", x, y);
case 'c' : x = x ^ y;
           y = x ^ y;
           x = x ^ y;
           PF ("%d, %d", x, y);
           break;
}
break;
```

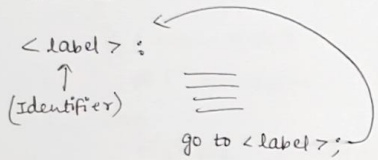
```
case 4 : PF (" select the options to perform Ascii operation \n
           a. upper case ⟷ lower case \n  b. char const ⟷ int const \n");
SF (" %c", &suboptions);
--fpurge (stdin);
PF ("Enter i/p \n");
SF ("%c", &k);
switch (suboptions)
{
  case 'a' : if ((k >= 65) && (k <= 90))
             {
               k = k + 32;
               PF ("%d", k);
             }
             else if ((k >= 97) && (k <= 122))
               k = k - 32;
               PF ("%d \n", k);
             }
             else
               PF ("Invalid i/p \n");
             break;
  case 'b' : if ((k >= 48) && (k <= 57))
             {
               k = k - 48;
               PF ("%d \n", k);
             }
             else
               PF ("Invalid i/p \n");
             break;
```

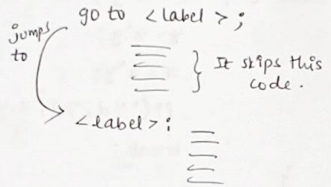
```
case 5 : PF ("Enter the i/p \n");
         SF ("%d", &t);
         x = 0;
         y = 1;
         for (i = 0; i < t; i++)
         {
           z = x + y;
           x = y;
           y = z;
           PF (" %d ", z);
         }
         break;
}
}
}
```

go to statement :-

⇒ goto stmt is a jump statement (control stmt) which changes cursor position.

Syntax :-





⊙ creates infinite loop if there is no proper cond".

⇒ label & goto stmt must be defined within the same fun^c.

⇒ When CPU executes this goto stmt, the cursor will jump to the location where the label is defined.

⇒ Even we can jump from the loop also.

⇒ This jump stmt must be used along with proper condition (If else (or) conditional operators).

Example :-

```
main()
{
 int num, sum=0;
 while(1)
 {
  SF("%d", &num);
  if(num==0)
  goto zero;
  sum=sum+num;
 }
 zero: PF("%d\n", sum);
}
```

⇒ when cond" is satisfied, it will immediately jump to zero label

⊛ Write a prgm to print 1 to 10 numbers by using goto stmt

⇒
```
main()
{
 int x;
 PF("enter the i/p\n");
 SF("%d", &x);
 print: printf("%d\n", x)
        x++;
        if(x<=10)
        goto print;
}
```

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

⊛ convert Decimal to Binary using goto stmt

⇒
```
main()
{
 int num, sum=0, i=1, rem, temp;
 PF("enter i/p\n");
 SF("%d", &num);
   temp=num
binary: rem=temp%2;
        temp=temp/2;
        sum=sum+rem*i;
        i=i*10;
        if(temp>0)
        goto binary;
 PF("%d\n", sum);
}
```

⊛ Assignment

⇒ print given range of Binary no's ⇒ print decimal no's using only goto stmt

I/p :- 1 & 1000

o/p
1 - 1
(2, 3, - - - 9) skip
10 - 2
11 - 3
(12, 13, - - - - 99) skip
100 - 4
101 - 5
(102 - - - - 109) skip
110 - 6
111 - 7
(112 - - - - 999) skip

1000 - 8.

```c
#include<stdio.h>
void main()
{
    int min, max, sum=0, i=1, temp, rem;
    PF("Enter the range min:max\n");
    SF("%d:%d", &min, &max);
    if(min>max)
        temp=min, min=max, max=temp;
start : sum=0;
        i=1;
        temp=min;
logic : rem=temp%10;
        if(rem>1)
        {min++;
         goto start;
        }
        temp=temp/10;
        sum=sum+rem*i;
        i=i*2;
        if(temp!=0)
        goto logic;
        PF("%d-%d\n", min, sum);
        min++;
        if(min<=max)
            go-to start;
}
```

✱ Functions :-

⇒ A function consist of group of 'c' stmts that defines a particular task.

⇒ The main prgm logic can be divided into to multiple functions.

✱ Advantages of using fun^c

① code size (prgm size) reduces
② prgm can be written in structured format.
③ understanding the flow of execution is easy.
④ Debugging is easy.
⑤ Less compilation errors will generate
⑥ Identification of errors also easy.
⑦ It will take less debugging time & Less modification time.
⑧ we can reUse that fun^c by defining in a library.

✱ Drawbacks :-

① compilation time increases
② performance degrade.

⇒ Each fun^c will have ① Fun^c declaration,
② Fun^c definition, &
③ Fun^c invocation (or) call.

⇒ whether it may be user defined fun^c (or) pre defined fun^c.

⇒ The pre defined fun^c declarations are present in header files.

⇒ The definitions are ~~also~~ present in libraries in pre compiled format.

⇒ Invocations we will do in c program.

⇒ In Kernal, we don't have any libraries.
. It has only kernal image.

⇒ Advantages of using libraries are it is ready made. It has already done 3 stages of compilation. we have to do only linking process.

⇒ User defined fun^c declarations, definitions & invocations we can see in our prgm.

⇒ fun^c name must be a meaningful name that signifies the specific task.

Syntax :-

<return type> <fun^c name> (, ,)

Formal arguments separated by comma operator.

(*) Formal arguments :- The variables which receives the i/p's during fun^c invocation time.

Fun^c definition :-
- Function header ⇒ <return type> <fun^c name> (, ,)
- Fun^c body { ≡ build the logic to perform the task }

⇒ (*) Fun^c definition (combination of fun^c header & fun^c body).

⇒ Decide the name based on functionality.

⇒ Construct the logics (Identify the series of steps to build the logic).

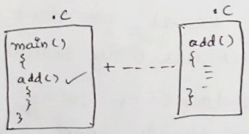
⇒ From there we will come to know the i/p's & return values.

⇒ Based on i/p's & return values we can define formal arguments & return values.

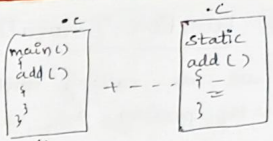
⇒ This fun^c definition we have to write within the same prgm.

⇒ Fun^c scope is global scope.

⇒ We can use the fun^c which is present in another .c file also.



⇒ We can restrict the usage by using static keyword.



⇒ NO storage class is applicable for formal arguments.

↓↓ NOT applicable

(*) Function declaration :-

Fun^c header ⇒ <return type> <fun^c name> (, ,); ⇒ Add semicolon to fun^c header to make it fun^c declaration

⇒ Fun^c declaration must be written on the top of the prgm after structures or Unions definitions.

⇒ Fun^c declaration will end with semicolon (fun^c header followed by semicolon).

⇒ Fun^c declarations are required for the compiler for the verification purpose in 2nd stage of compilation.

① ⇒ Fun^c name is verified in all the 3 locations. It will verify in ① definition, ② declaration & ③ Invocation.

② ⇒ No. of i/p's count are verified in all the 3 locations.

③ ⇒ I/p's & their types are verified in all the 3 locations.

④ ⇒ return values & their types are verified in 4 locations. (including return stmt).

⇒ Once these ④ verification's are done, declarations are automatically removed in 2nd stage of compilation.

⇒ If the verification is failed, it will stop the compilation process & generates compilation error.

(*) Function Invocation :-

⇒ We are calling that fun^c to perform that particular task by passing the valid i/p's.

- ⊙ We are calling fun^c invocation by
  - ↳ By using fun^c name
  - ↳ By passing valid i/p's.

19/10/22

⇒ When definitions are written before main() fun^c, declaration is not required. But when definitions are used after main fun^c then declarations are required.

⇒ return stmt will not only returns the value but also reads & tells the status of the fun^c.

⇒ If we want to terminate the fun^c (or) if we want to return from the fun^c we have to use return stmt.

⇒ Along with this return stmt, we can return only single value. It is not mandatory to always return a single value.

⇒ If we want to terminate the fun^c in b/w, we have to use return stmt with proper cond^n.

⇒ return stmt with value will signifies sometimes output of the task of fun^c, sometimes the status of the fun^c (succesfully executed or terminated in b/w).

⇒ Fun^c declaration is also known as fun^c prototype or fun^c signature.

⇒ If a fun^c is not returning a value after the task, the return type should be void.

⇒ exit (1) ⇒ exit with non-zero value signifies unsuccessful termination of the process in b/w without performing the task.
(or) exit (-1)

⇒ fun( ) ⇒ It can take 'n' no. of i/p's

⇒ fun(void) ⇒ It will not take/receive any i/p

⇒ If a fun^c is not receiving any i/p's mention the int type as void.

⇒ main fun^c will return its return value to the parent process.

⇒ Other than main fun^c, return values are returned to the invoked fun^c.

⇒ while writing the fun^c definitions we have to follow below 2 Mechanisms:

1. call by value (or) pass by value.
2. call by reference (or) pass by reference.

⇒ Fun^c invocation gets replaced with the return values after performing the task of fun^c.

⇒ return stmt in the fun^c will return the value to that particular fun^c.

⇒ return stmt in main fun^c will return the values to the main fun

⇒ exit (0) in any fun^c will terminates the process irrespective of fun^c.

①
```
int reverse_integer (int num)
{
  int temp, num, rem, sum = 0;
  while (temp != 0)
  {
    rem = temp % 10;
    temp = temp / 10;
    sum = sum * 10 + rem;
  }
  return sum;
}
```

②
```
int binary_decimal (int bin)
{
  int temp, bin, rem, dec = 0, i = 1;
  while (temp != 0)
  {
    rem = temp % 10;
    if (rem > 1)
      return -1;   ⇒ It will show the status of the fun^c.
    dec = dec + i * rem;
    i = i * 2;
    temp = temp / 10;
  }
  return dec;   ⇒ It will returns the o/p to the fun^c.
if (ret == -1)
  PF ("Invalid i/p \n");
else
  PF ("%d", dec);
}
```

```c
#include <stdio.h>
int reverse_integer (int);      // int reverse_integer (int num);
int binary_decimal (int);       // int binary_decimal (int bin);

void main (void)                → It will not take any i/p's
{                               (It will not return any value.)
    int num, opt, res, bin;
    while (1)
    {
      PF ("MENU\n 0.exit \n 1. reverse_int \n 2. bin_decimal \n");
      PF ("select the option \n");
      SF ("%d", &opt);
      Switch (opt)
      {
        case 0: exit(0);
        case 1: PF("enter the num\n");
                SF ("%d", &num);            ↖ 123
                res = reverse_integer (num);
                                  123
                                  321
                printf("%d\n", res);
                break;
        case 2: PF("enter binary i/p\n");
                SF ("%d", &bin);            ↖ 101
                res = binary_decimal (bin)
                                  101
                                  5
                if (res == -1)
                {Printf("invalid i/p\n");
                 break;
                }
                else
                   PF ("%d\n", res);
                break;
        default : PF ("invalid option\n");
      }
    }
}
```

⇒ compilation stages :-

⇒ The process of converting c stmts into machine instructions is known as compilation process. It has 4 stages:

1. pre processing stage
2. compiling "
3. Assembling "
4. Linking "

⇒ After completion of these 4 stages, it will generate an executable file with a default name a.out.

⇒ This executable file will have a file format :e., ELF.

① pre-processing stage :-

⇒ In pre-processing stage, pre-processor will process the 'c' stmts that starts with pre-processor directive.

⇒ In this stage, comments are removed. comments are not the part of executable file. But in debugging comments comments under executable file.

↳ which starts with #

- ⇒ inclusion of header files → # include <stdio.h>.
- ⇒ Macro definitions → # define MAX 5.
- ⇒ conditional compilation stmts → # if define
  # end if

⇒ Inclusion of header files stmts, # include <stdio.h> will get replaced with the content of the header file.

⇒ MACRO definitions are removed by substituting the content in the place of MACRO names.

⇒ conditional compilation stmts are removed by adding some 'c' codes or by deleting some lines of 'c' code.

⇒ This stage will generate pre-processed output which is represented with .i extension. This pre-processed o/p is processed 'c' code.

⇒ This o/p file is given as i/p to second stage i.e, compiling stage.

```
$ gcc -E ***.c > ***.i
```

⊙ It will stops & copies the info in .i file after the 1st stage of compilation.

⇒ This pre processed o/p file can be opened with the textual editor.

② Compiling Stage :-

⇒ In this stage compiler will converts 'c' stmts into to Assembly instructions.

⇒ This stage will generate o/p file with .s extension.

⇒ compiler & Assembly language are architecture dependent.

⇒ In this stage, fun^c declarations are verified with fun^c invocation & fun^c header.

⇒ If any parameter mis-matches it will generate compilation Error by stopping the compilation process.

⇒ Once declarations are verified successfully, they are removed.

⇒ Even structure definitions & union definitions are also verified with their members in appropriate locations.

⇒ Once verification is done successfully even they are also removed.

⇒ type def's gets substituted.

⇒ Now, it will verify the syntactical errors.

⇒ (when delimeters are not used at appropriate locations & if we not follow any one of the identifier rules in these case, it will generate syntactical errors).

⇒ Now, it will identify the symbols used in 'c' pgm.

[In this case, Global variables, static variables & fun^c names comes under symbols when we are discussing abt executable files]

and generates the corresponding assembly instructions to decide the memory sections for these symbols.

⇒ Now, it will start converting the 'c' Stmts into Assembling instructions.

⇒ For simple 'c' stmts, it will generate 1 or 2 Assembly instructions depends on operation.

⇒ For complex 'c' stmts, it will generates multiple Assembly instructions

⇒ Expressions are evaluated by using parse tree method.

⇒ In Assembly language also we will have fun^c definitions in Assembly code format.

c-code	Assembly-code
main()	main:
{	≡
≡	≡
≡	
}	

⇒ To stop the compilation after 2nd stage of compilation

```
$ gcc -S <cpgm file name>
```

generates

***.c ⇒ 1st & 2nd stage ⇒ ***.s file

***.i ⇒ 2nd stage ⇒ ***.s file.

⇒ To see the content of .s file we have to use normal textual editor (as it is in user understanding format).

③ Assembling Stage :-

⇒ This ***.s file is given as i/p to Assembling stage. so, Assembler will process this .s file & generates ***.o file. This Assembler is architecture dependent.

⇒ For every Assembly instruction it will generate corresponding opcodes & operands known as Mnemonics.

⇒ The op code is understable by that particular architecture CPU.

⇒ Operands can be data, memory locations or registers.

⇒ In this stage memory sections are formed & generates corresponding off set values.

⇒ For every instruction it will generate corresponding offset value.

⇒ $ gcc -c ***.c → 1st, 2nd & 3rd → generates ***.o
***.i → 2nd & 3rd → ***.o
***.s → 3rd → ***.o

⇒ To see the content of .o File & executable file we have to use special tools like objdump, readelf tools with various flags or options. (to convert binary to user understanding format).

④ Linking stage :-

⇒ The .o file is the exact replica of .c File.

⇒ Now this .o file is given as i/p to linking stage.

⇒ In this linking stage, the linker will combines the other multiple object files with this object file & generates a.out file.

⇒ These other object files contains the data specific to prgm loading & unloading and to generate the addresses for each instruction.

⇒ These other object files are commonly added for every 'c' prgm in linking stage.

⇒ when we are using pre-defined library fun^cs in our prgm we have to use -l flag followed by the library name to link the library.

⇒ when we link static library it will add the fun^c definitions which are called from our prgm.

⇒ when we link dynamic library or shared library just it will set the path of the library (size of executable file will not increase). Finally, it will generate a.out file.

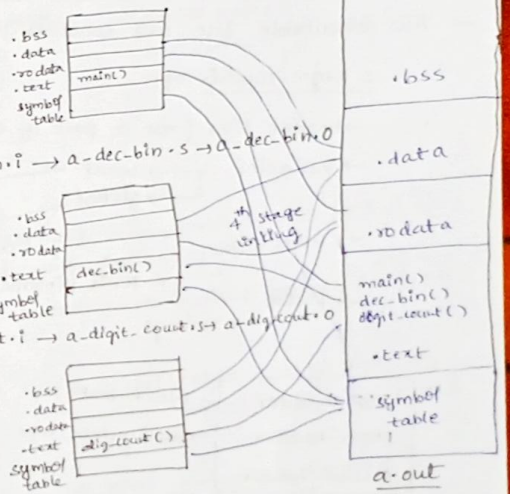
(*) NOTE :-

⇒ Debugging information can be added in 2nd stage of compilation when we use -g flag.

20/10/22

⇒ Sample.c → a-sample.i → a_sample.s → a_sample.o

only main()
{ ≡ }



⇒ To include all .c files use gcc *.c

int z=10; → global .data (D)

static int y = 30; → static global .data (d)

static int digit_count() → .text (t)
main() → .text (T)

⇒ A project consists of multiple .c files & header files to generate the single binary file we have to compile all these .c files together.

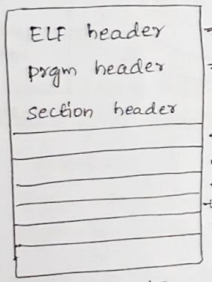
⇒ During the compilation process each .c file is processed separately up to 3rd stage & in 4th stage all these .o files are combined together & generates single binary file.

* Memory sections :- (or) ELF - file format

⇒ Executable file in linux will have ELF file format
⇒ This executable file will contains memory sections.

C prgm consists of

→ header files (not a part of executable file)
→ Variables → local → initialized / uninitialized
→ global → initialized / uninitialized.
→ functions → c stmts (instructions)
→ Data → strings.



⇒ File format will maintains its own header format).
⇒ Architecture info
...
• data section
• bss " (uninitialized section)
• rodata "
• text " (contains fun^c definitions in machine instructions format along with additional functions).
ro data (strings)
symbol table

ELF Format executable file.

⇒ The program which is under execution is known as process.
⇒ program will have memory sections.
⇒ process will have memory segments.
⇒ process will have 2 additional segments that are stack & heap
⇒ Executable file will not have stack segment & heap segment.
⇒ In executable file we have
① ELF header section
② prgm header "
③ section " "
④ .data section
⑤ .bss section
⑥ .rodata "
⑦ .text "
⑧ symbol table.

⑧ Symbol table :- It maintains symbols information, they are global variables, static variables & fun^c names.

<Off set address>	<section symbol>	<symbol name>
	text (t)	Fun^c name
	data (d)	variable "
	bss (b)	
	T	
	R	
	b	
	U	
	!	
	!	

⇒ If section symbols are represented with small letters (b,t,d etc) that means they are applied with static keyword.
⇒ If section symbols are represented with capital letters (D,T,R, etc) they are naml / general.

T/t → .text section
D/d → .data section
R/r → .rodata "
B/b → .bss "
U → Undefined

⇒ symbol table can be seen by using nm tool, obj dump & read elf tool.
⇒ nm <executable file name> or object file name.

⑦ .text section :-

⇒ This section contains fun^c definitions in machine instruction format

Interview Q/A

⇒ How do you get the base address of the fun^c.
(or)
⇒ How do you get the first instruction base address from the prgm.

(A) By using fun^c name [cuz fun^c name will always return the base address where the fun^c is defined].

⇒ .text section is the read only segment. we can't modify the content of text segment during the execution.

⇒ This text section is also known as code section. We can see the content of this text section by using obj dump (or) read ELF tools.

$ objdump -S executable filename
or
object file name.

⑥ rodata Section :-

⇒ It continues memory for the strings which are directly passed to the fun^c & the strings which are initialized by using pointers & also global variables which are applied with the constant keyword.

⇒ rodata segment is read only data segment.

⇒ If we try to modify the content of read only data segment it will generate run time error. i.e., segmentation fault error. So that means the process gets terminated there itself by displaying segmentation fault error.

⑤ .bss section (block started by the symbol).

⇒ This is known as uninitialized data section. This section contains only info about uninitialized global variables.

⇒ This section memory gets allocated when you load the pgm. (.bss segment).

④ .data section :-

⇒ This contains memory for initialized global variables & static variables.

(*) NOTE :-

* Uninitialized static variables memory is allocated in .bss segment

⇒ Both .bss & .data segment will have read & write permissions.

③ Section header :-

⇒ It contains or maintains each section information

② program header :-

⇒ It maintains offset address, process loading info, byte order storage format, Architecture dependency info.

① ELF header :-

⇒ It contains ELF File Format info.

21/10/22

⇒ $ size is a tool which will displays each memory section size of each executable file.

$ size *** → executable file name along with path.

displays *** Text (decimal) data (deci) bss (dec) overall size (decimal/Hexa decimal).

```
int addition (int, int);
main()
{
  int a=10, b=20, res=0;
  printf("main : %p : a → %p - %d\n", main, &a, a);   (1st instruction machine code)
  PF("main : %p : b → %p - %d\n", main, &b, b);
  PF("main : %p : res → %p - %d\n", main, &res, res);
  PF("main : calling addition\n");
  res = addition(a, b);        (2 operations: ① Func call ② assignment)
                  10  20
                  30
}

int addition(int a, int b)
{
  PF("addition: %p : a → %p - %d\n", addition, &a, a);
  PF("addition: %p : b → %p - %d\n", addition, &b, b);
  return (a+b);   (addition returns the value)
}
```

o/p a.out

main : <virtual add> : a → 0x1000 – 10

main : < > : b → 0x1004 – 20

main : < > : res → 0x1008 – 0

main : calling <addition>.

addition : < > : a → 0x2000 – 10

addition : < > : b → 0x2004 – 20

main : < > : res → 0x2008 – 30

Assignment

① construction of libraries (static & dynamic) for 2 or 3 functions.

② write commands to verify the memory sections using 2 tools
objdump
readelf

③ write one application to verify the functionality and also check each & every compilation o/p.

① < header files.h >
int count_0 and 1 (int)
int dec_binary (int)
int sum_of_digits (int)

count_0 and 1.c
int count_0 and 1 (int).
{
int a=0, k=0, bit;
for (bit = 0; bit <= 31; bit++)
{
if ((n & (0x1<<bit)) == 0)
a++;
else
k++;
}
PF("zeros --- %d ones --- %d\n", a, k);
}

sum_of_digits.c
sum_of_digits (int)
int temp, sum = 0, rem;
temp = num;
sum = 0;
while (temp > 0)
{
rem = temp % 10;
temp = temp / 10;
sum = sum + rem;
}
PF("\nsum = %d", sum);
return sum;
}

header files.h

dec_binary.c
int decimal_binary (int)
{
int bit;
for (bit = 31; bit >= 0; bit--)
((n & (0x1<<bit)) == 0) ? PF("0") : PF("1");
PF("\n");
printf ("\n");
return n;
}

main.c file | final.c

#include <stdio.h>
#include "headerfiles.h"
void main()
{
int opt, n, res, num;
PF("\n menu \n 1. count 0 and 1 \n 2. decimal_binary \n 3. sum_of_digits \n");
SF("%d", &opt);
--fpurge (stdin);
switch (opt)
{
case 1: PF("\n enter i/p\n");
SF("%d", &n);
res = count_0_and_1(n);
PF("\n o/p\n", res);
break;

```
case 2 : PF("enter i/p\n");
         SF("%d", &x);
         res = decimal_binary(x);
         printf("%d\n", res);
         break;
case 3 : PF("enter i/p\n");
         SF("%d", &num);
         res = SumOf_digits(num);
         PF("%d\n", res);
         break;
    }
}
```

⇒ Now to generate static library & dynamic library we have to create .o files of fun^c files by stopping the compilation after third stage.

$ gcc -c count_oandi.c dec_bin.c sumof_digits.c

⇒ we will get .o files of three fun^c declaration files.

⇒ Now to generate static library.

$ ~~gcc~~ $ ar rcs libstatic.a count_oandi.o dec_bin.o sumof_digits.o

- libstatic : static library name
- a : archieve
- .o files

- static library is generated/created.

⇒ Now to generate dynamic library

compile all .c files to get .o files.

gcc -c -FPIC count_oandi.c dec_bin.c sumof_digits.c

- -c : To stop compilation after 3rd stage & to create .o file
- -FPIC : to make code position independent
- .c files

⇒ To create dynamic library

gcc -shared -o libdynamic.so *.o

- libdynamic : library name
- .so : shared object (dynamic library)
- *.o : all .o files

⇒ To verify whether it is created or not

nm -D libdynamic.so

⇒ Now to link static & dynamic library

$ gcc -L. 3in1.c -l static

gcc -L. 3in1.c -l dynamic

- 3in1.c : main file name

⇒ To see section address

objdump -S a.out ⇒ To verify through Assembly instructions

objdump -s a.out ⇒ To verify memory section details (text segment)

⇒ To Export dynamic libraries

~~gcc~~ $ export LD_LIBRARY_PATH = /engineer@70dell:~~/home~~/engineer/Desktop :$LD_LIBRARY_PATH

Path

26/10/2022

⇒ The variables which passes the inputs to formal arguments during the func invocation time they are known as Actual arguments.

func call (or) invocation ⇒ scanf("%d-%d", &x, &y);

&x, &y → through address (call by reference values)

↓ actual arguments.

int add(int, int)

↳ formal arguments.

✱ In which scenario stack grows downwards ?

⇒ ① when a function is invoked. (Block of memory is created when func is called in stack frame)

② when a new thread is created.

③ when we request the memory by using alloc✱() func.

④ when we use reclusive functions in our program.

✱ Explain the scenarios where heap grows upwards ?

request the memory

① when we use DMA calls like malloc(), calloc(), realloc(), functions

② when we request the memory by using system calls sbrk() brk()

③ when we use mmap functions (maping)

④ when we request shared memory by mmap().

⇒ when heap or stack tries to overwrite each other, Kernal will not allow this operation they will kill that application by Sending a signal SIGSEGV (kernal) & the process gets terminated by displaying a run time error by displaying the error msg i.e., segmentation fault.

⇒ Every process will have one thread i.e., main thread. The entry point of main thread is main() func.

⇒ for every thread internally it will allocate a stack area. That stack area is pointed by stack pointer.

⇒ program counter (PC), register will points (contains) the next instruction address

⇒ In call by value mechanism, we cant update the content on actual arguments

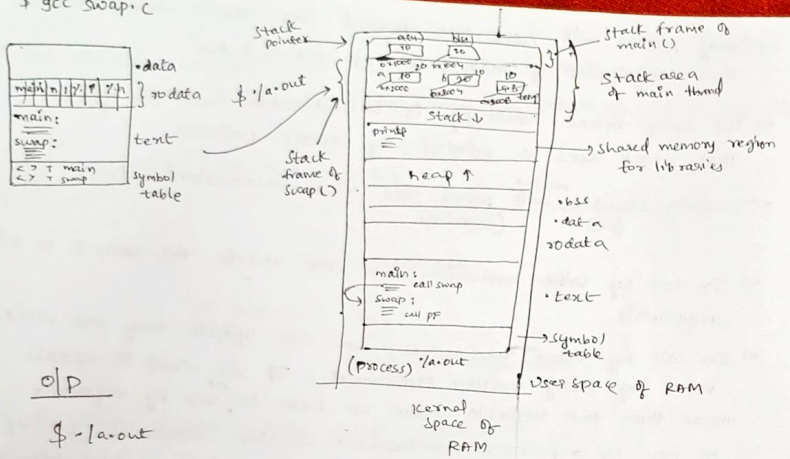
⇒ In call by value mechanism, we can update only one variable value by using return statement. If we want to update more than one variable value we have to use call by reference or pass by reference mechanism. In this mechanism (call by reference) we will pass memory location addresses or variable's addresses as actual arguments. To receive these addresses we use pointers as formal arguments.

swap.c

```
main()
{
  int a=10, b=20;
  PF("main: a: %p - %d\n", &a, a);
  PF("main: b: %p - %d\n", &b, b);
  swap(a,b)
  Printf("main: a: %p - %d\n", &a, a);
  PF("main: b: %p - %d\n", &b, b);
}
```

```
void swap(int a, int b)
{
  int temp;
  PF("swap: a: %p - %d\n", &a, a);
  PF("swap: b: %p - %d\n", &b, b);
  temp = a;
  a = b;
  b = temp;
  PF("swap: a: %p - %d\n", &a, a);
  PF("swap: b: %p - %d\n", &b, b);
}
```

$ gcc swap.c



o/p

$ ./a.out

main : a : 0x1000 – 10
main : b : 0x1004 – 20
swap : a : 0x2000 – 10
swap : b : 0x2004 – 20
swap : a : 0x2000 – 30
swap : b : 0x2004 – 10
main : a : 0x1000 – 10
main : b : 0x1004 – 20

⇒ Pointers :-

⇒ pointers are used to access the data with the help of memory location address.

⊛ Advantages :-

① To access the data with the help of address.
② To implement call by reference mechanism in functions.
③ To construct array concept (Arithmetic operations on pointers)
④ To access dynamically requested memory.
⑤ Pointer concept + structures concept we can develop Data structures (Arrays comes under Data structures).

⇒ within that process address space if we know the address we can access the data by using pointer variable. Address doesn't have any scope. But, pointer variables will have scope based on their declaration.

⇒ Like, normal variables same life time, scope, default values, Storage ~~applications~~ locations are applicable for pointer variables as well.

⇒ when you deal with the pointers, it will come across the runtime errors frequently.

⇒ Pointers must contain valid address memory location.

⇒ Invalid address means the address where we don't have valid data or it can be out of boundary address.

⇒ we can differentiate a pointer variable from a normal variable by dereferencing operator * x (or) * ptr (defaultly use this)

```
main()
{
int x = 10;
}
```
↑ Initialized Normal Local Variable

```
main()
{
int *x = &y;
}
```
↑ Initialized local integer pointer variable.

⇒ * ptr
↑
dereferencing operator / indirectional operator / astric symbol / star.

⊛ Steps to be followed to access the data by using pointer variable.

Step:- 1 :- Declare (or) define a pointer variable based on data type `int *iptr;`

Step:- 2 :- Assign the address of the memory location to the pointer which you want to access `iptr = &x;`

Step:- 3 Dereference the pointer to access the data from that address `*ptr = 20;`

⇒ when we dereference the pointer we will get the data present at that address.

Pointer size doesn't depends on data type. It depends upon architecuture machine type.

Machine	
32 bit	⟶ 4 bytes
64 bit	⟶ 8 bytes.

⇒ A pointer which contains gasbage value (invalid address) it is known as wild pointer.

⇒ A pointer which contains null or zero as the address it is known as Null pointer.

```
ptr = 0;   (Generally we don't use this)
ptr = NULL;
      ⇓
    Macro
```

⇒ void pointer

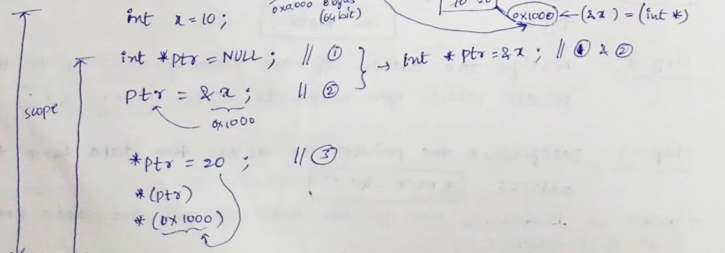
⇒ Dangling pointer.

⇒ we must update a pointer with NULL when it is not pointing any valid memory location. This NULL acts as reference.

(like while(1) while(x>0) → reference point)

⇒ Importance of data type in pointer Variable :-

⇒ It signifies or it tells how many bytes to be accessed from the stored address when you dereference the pointer.



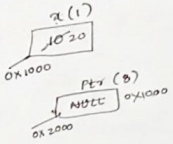

```
printf("%p - %d - %d - %d\n", &x, x, size of(&x), size of(x));
       0x1000   20   8   4        (int *)   (int)

printf("%p - %d\n", ptr, size of(ptr));
       0x1000   8

printf("%d - %d\n", *ptr, size of(*ptr));
       20   4

printf("%p - %d\n", &ptr, size of(&ptr));
       0x2000   8
```

⇒ char *
short int *
Float ~~int~~ *
double *



⇒
```
char x = 10;
char *ptr = NULL;
ptr = &x;
*ptr = 20;
```

```
PF("%p - %d - %d - %d\n", &x, x, size of(&x), size of(x));
   0x1000   20   8   1        (int *)

PF("%p - %d\n", ptr, size of(ptr));
   0x1000   8

PF("%d - %d\n", *ptr, size of(*ptr));
   20   1

PF("%p - %d\n", &ptr, size of(&ptr));
   0x2000   8
```

Pointer size doesn't depends on data type. It depends upon architecture machine type.

Machine

32 bit ⟶ 4 bytes

64 bit ⟶ 8 bytes.

⇒ A pointer which contains garbage value (invalid address) it is known as wild pointer.

⇒ A pointer which contains null or zero as the address it is known as Null pointer.

ptr = 0; (Generally we don't use this)

Ptr = NULL;

⇓ MACRO

⇒ void pointer

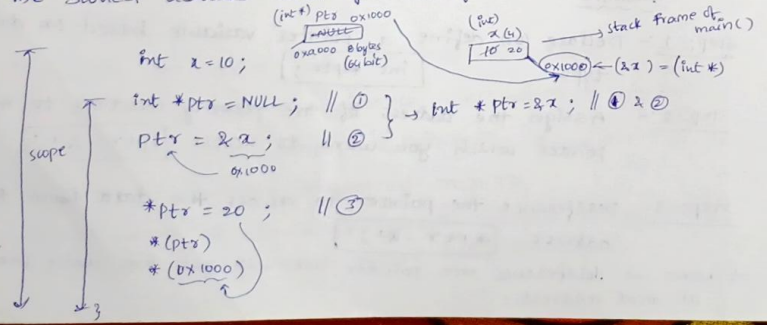
⇒ Dangling pointer.

⇒ We must update a pointer with NULL when it is not pointing any valid ~~add~~ memory location. This NULL acts as reference.

(like while(1!=0)
while(x>0) → reference point)

⇒ Importance of data type in pointer variable :-

⇒ It signifies or it tells how many bytes to be accessed from the stored address when you dereference the pointer.



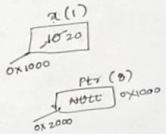
printf("%p - %d - %d - %d\n", &x, x, sizeof(&x), sizeof(x));
0x1000 20 8 4 (int*) (int)

printf("%p - %d\n", ptr, sizeof(ptr));
0x1000 8

printf("%d - %d\n", *ptr, sizeof(*ptr));
20 4

printf("%p - %d\n", &ptr, sizeof(&ptr));
0x2000 8

⇒ char *
short int *
float int *
double *



⇒ char x = 10;
char *ptr = NULL;
ptr = &x;
*ptr = 20;

PF("%p - %d - %d - %d\n", &x, x, sizeof(&x), sizeof(x));
0x1000 20 3 1 (int*)

PF("%p - %d\n", ptr, sizeof(ptr));
0x1000 8

PF("%d - %d\n", *ptr, sizeof(*ptr));
20 1

PF("%p - %d\n", &ptr, sizeof(&ptr));
0x2000 8

⇒ float x = 10;

float *ptr = NULL;

ptr = &x;

*ptr = 20;

Pf("%p - %f - %lf - %ld\n", &x, x, sizeof(&x), sizeof(x));
(%p: 0x1000, %f: SD=00000, %lf: 8, %ld: 4)

PF("%p - %ld\n", ptr, size of (ptr));
(%p: 0x1000, %ld: 8)

PF("%f - %ld\n", *ptr, size of (*ptr));
(%f: 20, %ld: 4)

PF("%p - %ld\n", &ptr, size of (&ptr));
(%p: 0x2000)

⇒ double x = 10;

double *ptr = NULL;

Ptr = &x;

*ptr = 20;

PF("%p - %lf - %d - %ld\n", &x, x, sizeof(&x), sizeof(x));
(%p: 0x1000, %lf: 20, %d: 8, %ld: 8)

PF("%p - %d\n", ptr, size of (ptr));
(%p: 0x1000, %d: 8)

Pf("%lf - %lf\n", *ptr, size of (*ptr));
(%lf: 20.0000, %lf: 8)

PF("%p - %d\n", &ptr, size of (&ptr));
(%p: 0x2000, %d: 8)

* NULL - MACRO Definition

⇒ Used with null-pointer operations & fun^{c}.

⇒ Null defined in header files: CRT DBG.H, LOCALE.H, STDDEF.H, STDIO.H, STDLIB, TCHAR.H, TIME.H, WCHAR.H.

⇒ The C library MACRO NULL is the value of a null pointer constant. It may be defined as ((void*)0), 0 or 0L depending on compiler vender

27/10/22

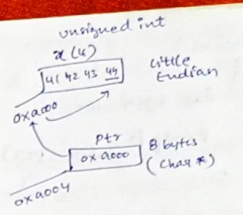
⇒ unsigned int x = 0X41424344;

unsigned char *ptr = &x;

PF("%x - %d - %c", *ptr, *ptr, *ptr);

0x44 — 68 — D

(When we pass address of the data it access only 1 byte even if it %x format specifier)



0100 0100 = 64+4 = 68 = 'D'

0xa000 0xa004

⇒ void Swap (int *ptr1, int *ptr2)

{

int temp;

temp = *ptr1; ⇒ temp = *(0x1000) = 10

*ptr1 = *ptr2; *(0x1000) = *(0x1004) = 20 (indirectly)

*ptr2 = temp; *(0x1008) = temp = 10 b (indirectly)

}

main()

{

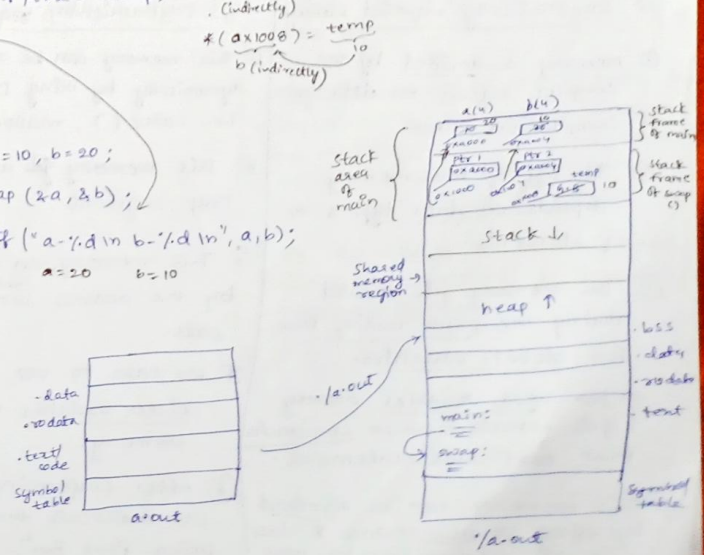
int a = 10, b = 20;

Swap(&a, &b);

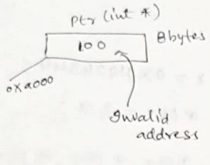
Printf("a - %d \n b - %d \n", a, b);

a = 20 b = 10

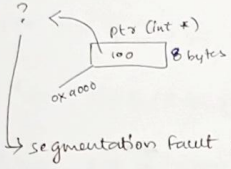
}



⇒ main()
{
int *ptr = 100;
printf("%d\n", ptr);
}
↳ 100



⇒ main()
{
int *ptr = 100;
PF("%d\n", *ptr);
}



↳ segmentation fault error.

Dynamic Memory allocation calls (DMA calls)

⇒ Differences b/w statically requested memory & dynamically requested memory :-

⇒ Statically requested memory	Dynamically requested memory
① Memory is decided by the compiler based on data type at compilation time	① This memory can be requested dynamically by using DMA calls. i.e., calloc(), malloc().
② The size of this memory depends on data type & no. of elements.	② This memory gets allocated in heap segment.
③ This memory gets allocated during the pgm loading time for global variables. ⇒ For local variables memory gets allocated when CPU executes that particular statement.	③ This memory can be accessed by the address returned by DMA calls.
④ This memory can be accessed by using variable name & also with its address (variable's base address).	④ We have to use dedicated pointer variables to access that memory.
	⑤ After performing the task, we can deallocate that memory by using free fun^c.

⑤ This memory gets deallocated
* for local variables → when fun^c (or) block completes.
* for global variables → when pgm terminates.

⑥ These variables will have scope. They are valid within the fun^c or file scope or block scope.

⑦ ⇒ Local variables memory gets allocated in stack frame of corresponding fun^c.
⇒ Global variables memory gets allocated in bss or data segment

⑧ Statically requested memory, neither we can increase or decrease nor dellocate at run time.

⑨ Here, we can't utilize this memory effectively & efficiently.

⑤ This memory size can be increased or decreased by using realloc() during runtime.

⑥ This dynamically requested memory doesn't depends upon data type. We can store any type of data in this memory based on our requirement.

⑦ If we know the address of this memory we can access this from any function.

⑧ Dynamically requested memory doesn't have any scope. we can access them within that address space.

⑨ Improper handling of this memory may leads to memory leaks.
⇒ The developer has to take care of these memory leaks. To find out these memory leaks, valgrind tool is used

⑩ After deallocating the memory by using free(), the pointer contains the address where we don't have valid memory, that pointer is known as Dangling pointer

⑪ We can utilize the memory more effectively & efficiently by Dynamically requested memory

⇒ Differences between calloc() & malloc()

① → malloc() will allocate the memory as a single block
→ calloc() will " " " block by block continuously (or) 'n' no. of blocks.

② → malloc() will take the single i/p i.e., overall size of the block
→ calloc() will take 2 inputs i.e., No. of blocks & block size.

③ → malloc() will allocate the memory as it is with garbage values.
→ calloc() will " " " by updating with zeroe's (0).

④ → malloc() is faster than calloc().
→ calloc() will internally invokes ~~two~~ functions (memset functions).

⇒ These malloc(), calloc(), realloc(), library functions internally invokes brk() or sbrk() system calls.

⇒ These DMA calls we have to include stdlib.h header file.

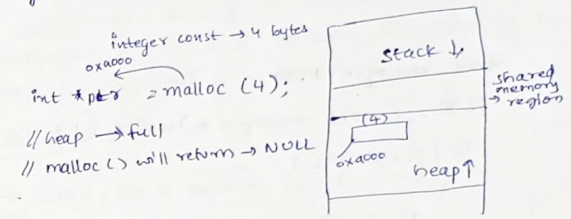
⇒ void pointer is known as generic pointer. (eg:- void *ptr;)

```
Void * malloc ( size_t size);
                  ↑
            type def of
            int type
```

⇒ This malloc() will take the single input i.e., no. of bytes as a size.

⇒ The memory requested by malloc() gets allocated in heap segment & it will return the starting base address.

⇒ malloc() on failure in allocating the memory it will return null. Dereferencing null we will get segmentation error



⇒ when heap segment is full, (if) when we are requesting the memory by using malloc(), calloc(), realloc() these functions will return null on failure

Verify the return values of these DMA calls if it

⇒ If it is NULL, terminate the application by calling exit() to avoid runtime errors.

```
if(ptr == NULL)
{
  PF(" Fail to allocate memory in heap segment \n");
  exit(1);
}
```

exit(1); → Non-zero value.

syntax for malloc()

DMA (FULL)

returns add 0xa000

For integer constant = (4 bytes) 10

```
int * ptr = malloc (4);   (It is not a standard way of writing the statement)
```

```
int * ptr;
0xa000
Ptr = (int *) malloc (1 * sizeof (int));   (standard type (or) way)
if (ptr == NULL)
{
  PF(" Fail to allocate memory in heap seg \n");
  exit(1);
}
PF("%p - %d \n", ptr, *ptr);
   0xa000      10
free (ptr); // mandatory   ** To deallocate the memory after the execution.
}
```

```
struct employee
  {

  }
struct employee *ptr;
ptr = (struct employee *) malloc (1 * sizeof (struct employee));
if (ptr == NULL)
  {
    PF(" ");
    exit(1);
  }
```

Syntax for free ( ) :-

```
void free (void *ptr);
```

⇒ By using this free function, we can deallocate the memory which is requested dynamically (malloc(), calloc(), realloc()).

⇒ This Fun^c will take single input i.e., the address returned by malloc or calloc or realloc functions.

⇒ This fun^c will not return anything.

⇒ After deallocating the memory, the pointer contains the address where the memory does not belongs to process address space. That pointer is known as Dangling pointer. We should not dereference the dangling pointer. (don't access the memory from that address from the process).

⇒ Dangling pointers contains invalid address.

28/10/22

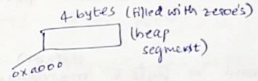
⇒ Calloc ( ) :-

⇒ This function will allocate the memory at run time in heap segment as a continuous blocks and returns the starting base address that memory is initialized with zero's.

(*) Declaration of calloc ( )

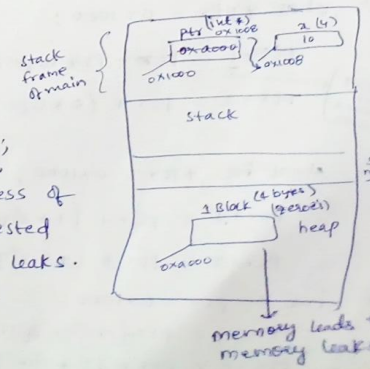
```
void * calloc (size_t nmem, size_t size);
int * ptr = (int *) calloc (1, sizeof (int));
```

(0xa000 returned; 1 = No. of elements; sizeof(int) = each element size.)



⇒ main()
```
{
  int x=10;
  int *ptr= (int *) malloc (1 * sizeof (int));
  int * ptr = (int *) calloc (1, sizeof (int));
  if (ptr == NULL)
    {
      PF(" failed to allocate memory \n");
      exit(1);
    }
  // ptr = &x;
  *ptr = 10;
  PF("%p - %d\n", ptr, *ptr);
  free (ptr);
}
```

(0xa000 returned; 1 = no. of elements; sizeof(int) = each element size; ptr = &x; 0x1008)



(*) NOTE :-

As soon as we update base address of variable x, the dynamically requested memory may leaks to memory leaks. (0xa000) till the prgm terminates.

⇒ The 4 bytes memory at address 0x a000 cannot be accessed by current process or by other processes till process gets terminated.

⇒ we must use dedicated pointers for dynamically requested memory throughout the process until we deallocate that memory or process termination.

* Arithmetic operations on pointers :-

⇒ If we have 'n' bytes of data, i want to access the data byte by byte we have to use character pointer (char *ptr) with arithmetic operations

⇒ If we want to access same 'n' bytes (data) memory → 2 bytes, 2 bytes we have to use short int pointer with arithmetic operations.

⇒ If we want to access same 'n' bytes of data 4 bytes 4 bytes from a given address, we have to use integer pointer with arithmetic operations.

⇒ when we increment character pointer it will jump 1 byte
⇒ when we increment short int pointer " " " 2 bytes.
⇒ " " " integer pointer " " " 4 bytes.

⇒ char *ptr = 0x1000;

ptr+1 = ptr + (1 * size of (char)) = 0x1000 + (1*1) => 0x1001
ptr+2 = ptr + (2 * size of (char)) = 0x1000 + (2*1) ⇒ 0x1002

(Index or reference values)

⇒ short int *ptr = 0x1000;

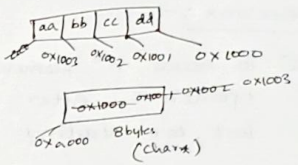
ptr+1 = ptr + (1 * size of (short int)) = 0x1000 + (1*2) ⇒ 0x1002
ptr+2 = ptr + (2 * size of (short int)) = 0x1000 + (2*2) ⇒ 0x1004

⇒ int * ptr = 0x1000;

ptr+1 = ptr + (1 * size of (int)) = 0x1000 + (1*4) => 0x1004
ptr+2 = ptr + (2 * size of (int)) = 0x1000 + (2*4) ⇒ 0x1008

Ex:-
⇒ If we have 2000 bytes memory, we have stored 500 Elements information, If we want to access 100th element info we have to write ptr+(100-1)

⇒ unsigned int x = 0xaabbccdd;
unsigned char *ptr = &c;



Printf ("%p - %x\n", ptr, *ptr);
0x1000 - dd
ptr++; // ptr = ptr+1;
Printf ("%p - %x\n", ptr, *ptr);
0x1001 bb
ptr++; // ptr = ptr+1;
printf ("%p - %x\n", ptr, *ptr);
0x1002 bb
ptr++;
Printf ("%p - %x\n", ptr, *ptr);
0x1003 aa

⇒ This method is not recommended

instead use ⇒ 4 methods

① ptr[i]
② i[ptr]
③ *(ptr+i)
④ *(i+ptr)

(ptr+0), → *(ptr+0)
(ptr+1), → *(ptr+1)
(ptr+2), *(ptr+2)
(ptr+3), *(ptr+3)

char *ptr;
ptr = 0x1000;
PF("%p\n", ptr++); → 0x1001
ptr = 0x1000;
PF("%p\n", ++ptr); ⇒ 0x1001

⇒ In arithmetic operation of pointers the first method is not applicable recommendable. *ptr, ptr++ } Not recommendable

Reasons :-

① To access n^{th} element, we have to perform (n-1) arithmetic operations. After performing those many operations we lost the starting base address.

② We can access the elements only in forward direction.

③ This method will degrade the performance of CPU. There is is a possibility of memory leaks when you deal with dynamic memory.

④ Reaccessing the elements from the begging is not possible in this method.

⇒ Apart from this method we can follow other 4 methods.

To access data/elements

2nd method = *(ptr+i)

III " = *(i+ptr)

IV " = ptr[i] = * square braces represents dereferencing → *(ptr+i)

V " = i[ptr] → = *(i+ptr)

(IV, V: accessing elements by using array method)

To access address

I = ptr

II - (ptr+i)

III - &ptr[i] ⇒ &(*(ptr+i))

IV - &i[ptr] ⇒ &(*(i+ptr))

5 integers: 10, 20, 30, 40, 50 (20 bytes)

```
main()
{
  int *ptr;
  int i;
// ptr = (int *) malloc(5 * sizeof(int));
  ptr = (int *) calloc(5, sizeof(int));
  if(ptr==NULL)
  {
    printf("Failed to allocate memory in heap\n");
    exit(1);
  }
ptr[0] = 10;
ptr[1] = 20;
ptr[2] = 30;
ptr[3] = 40;
ptr[4] = 50;
```

(or)

```
scanf("%d", &ptr[0]);
scanf("%d", &ptr[1]);
scanf("%d", &ptr[2]);
scanf("%d", &ptr[3]);
scanf("%d", &ptr[4]);
```

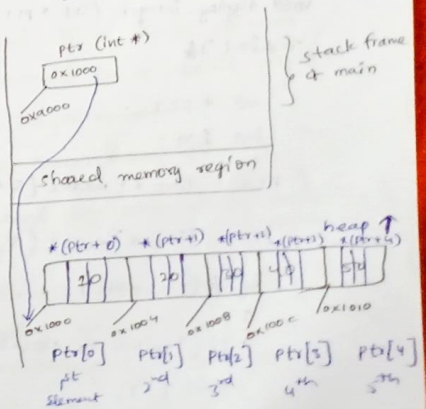
ptr[0] = *(ptr + (0 * sizeof(int)))
= *(0x1000 + 0) ⇒ *(0x1000) — 4 bytes

ptr[1] = *(ptr + (1 * sizeof(int)))
*(0x1000 + 4) ⇒ *(0x1004) — 4 bytes

ptr[2] = *(ptr + (2 * sizeof(int)))
*(0x1000 + 8) ⇒ *(0x1008) — 4 bytes

ptr[3] = *(ptr + (3 * sizeof(int)))
*(0x1000 + (3*4)) ⇒ *(0x100C) — 4 bytes
12 = 0xC

ptr[4] = *(ptr + (4 * sizeof(int)))
*(0x1000 + (4*4)) ⇒ *(0x1010) — 4 bytes
16 = 0x10



```
for (i=0 ; i<5 ; i++)        <- 5 elements
 {
   printf ("enter the i/p \n");
   scanf ("%d", &ptr[i]);
 }

   for (i=0 ; i<5 ; i++)
II nd ⇒ PF ("%d's element → %p - %d\n", i+1, ptr+i, *(ptr+i));
III  ⇒ PF ("%p - %d\n", i+ptr, *(i+ptr));
IV   ⇒ PF ("%p - %d\n", &ptr[i], ptr[i]);
V    ⇒ PF ("%p - %d\n", &i[ptr], i[ptr]);

    free (ptr);
 }
```

Using functions

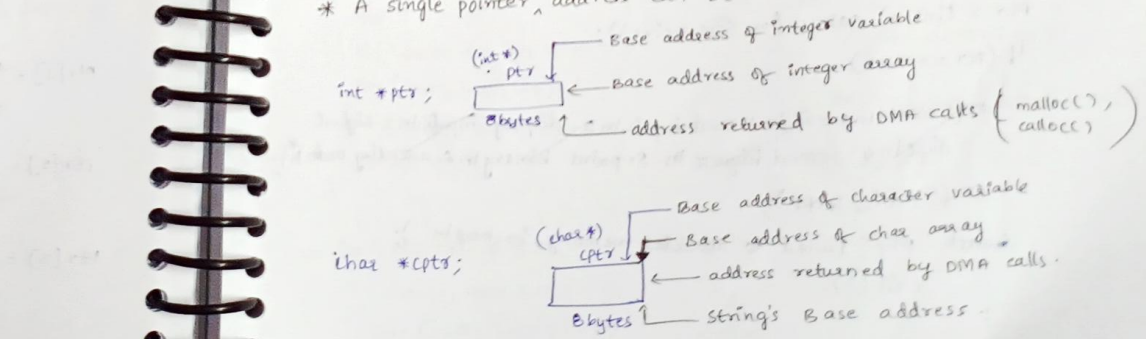
```
#define MAX 5
int read_integer (int *ptr, int n);
Void display_integer (int *ptr, int n);
main()
 {
   int *ptr;
   int i,n;
// Ptr = (int *) malloc (5 * size of (int));
   ptr = (int *) calloc (5, size of (int));
   if (ptr == NULL)
    {
      PF ("failed to allocate memory in heap\n");
      exit (1);
    }
n = read_integer (ptr, 5);      // ptr: 20 bytes address (5 elements), 0x1000; 5: No. of elements
   display_integer (ptr, n);
   free (ptr);
 }
```

(ptr, MAX)

(ptr, MAX)

```
int read_integer (int *ptr, int n)        // ptr: 0x1000, n: 5
 {
   for (i=0 ; i<n ; i++)
    {
      PF ("enter the i/p\n");
      SF ("%d", &ptr[i]);
    }
    return i;
 }
void display_integer (int *ptr, int n)
 {
   for (i=0 ; i<n ; i++)
    {
I   PF ("%d's Element → %p - %d\n", i+1, ptr+i, *(ptr+i));
II  PF ("%p - %d\n", i+ptr, *(i+ptr));
III PF ("%p - %d\n", &ptr[i], ptr[i]);
IV  PF ("%p - %d\n", &i[ptr], i[ptr]);
    }
 }
```

* A variable address can be stored in a single pointer.
* A single pointer address can be stored in a double pointer.



29/10/2022

Task

0. exit
1. read i/p's
2. display i/ps
3. biggest value
4. 2nd biggest value
5. Binary
6. ascending order

```
#include <stdio.h>
#include <stdlib.h>
int read_inputs (int *ptr, int n);
int display_inputs (int *ptr, int n);
int highest_digit( int *ptr, int n);
int second_highest ( int *ptr, int n);
int ascending_order (int *ptr, int n);
int Print_binary (int *ptr, int n);
int main()
{
  unsigned int n, *ptr, opt;
  PF("enter n value (no. of elements)\n");
  SF("%d", &n);
  Ptr = (int *) calloc (n, size of(int));
  if (ptr == NULL)
  {
    PF("MENU \n0. exit \n1. read inputs \n2. display inputs \n3. highest
    digit \n4. second highest \n5. print binary \n6. ascending order\n");
    SF("%d", &opt);
    Switch  PF("failed to allocate memory in heap\n");
        exit(1);
  }
while (1)
{
  -- fpurge(stdin);
  PF("MENU\n0. exit\n1. read inputs\n2. display inputs\n3. highest
element digit\n4. second highest\n5. print binary\n6. ascending order\n");
  Scanf("%d", &opt);
  switch (opt)
  {
    case 0: exit(0);
    case 1: printf("enter n values \n");
            read_inputs (ptr, n);
            break;
    case 2: PF("displaying the i/p's \n");
            display_inputs (ptr, n);
            break;
    case 3: highest_digit (ptr, n);
            break;
    case 4: second_highest (ptr, n);
            break;
    case 5: print_binary (ptr, n);
            break;
    case 6: ascending_order (ptr, n);
            break;
    default : printf("Invalid option\n");
  }
}
  free (ptr);
}
int read_inputs (int *ptr, int n)
{
  int i;
  PF("enter the i/p's \n");
  for (i=0; i<n; i++)
  SF("%d", &ptr[i]);
  return 0;
}
int highest_digit (int * ptr, int n)
{
  int i, max = 0;
  for (i=0; i<n; i++)
  {
    if (max < ptr[i])
      max = ptr[i];
  }
  PF("highest digit is : %d\n", max);
return 0;
}
int display_inputs (int *ptr, int n)
{
  int i;
  for (i=0; i<n; i++)
  {
    PF("%p - %d\n", &ptr[i], ptr[i]);
  }
  return 0;
}
```

④ ⇒
```
int second_highest (int *ptr, int n)
{
    int big=0, i=0, _2big=0;
    for (i=0; i<n; i++)
    {
        if (ptr[i]>big)
        {
            _2big = big;
            big = ptr[i];
        }
        Else if (ptr[i] > _2big)
            _2big = ptr[i];
    }
    PF("second highest = %d\n", _2big);
}
```

⑤
```
int paint_binary (int *ptr, int n)
{
    int i, j=1, rem=0, sum=0, temp;
    for(i=0; i<n; i++)
    {
        temp = ptr[i];
        sum=0;
        rem=0;
        j=1;
        while (temp>0)
        {
            rem= temp % 2;
            temp = temp /2;
            sum= sum + rem*j;
            j = j*10;
        }
        PF ("%d-%d\n", ptr[i], sum);
    }
}
```

⑥
```
int ascending_order (int *ptr, int n)
{
    int i, j, temp;
    for (i=0; i<n; i++)
    {
        for (j=i+1; j<n; j++)
        {
            if (ptr[i] > ptr[j])
            {
                temp = ptr[i];
                ptr[i] = ptr[j];
                ptr[j] = temp;
            }
        }
        PF("%d\n", ptr[i]);
    }
}
```

31/10/2022

⑧ Realloc ( ) :-

→ on success it will return same add
→ on failure it will return NULL

```
void * realloc (void *ptr, size_t size);
```
(ptr: starting address of memory requested by malloc(), calloc(); size_t: new overall size)

⇒ This function is used to increase or decrease the size of dynamically requested memory by using malloc, calloc fun^{cs} at same address with new size.

⇒ For realloc ( ) also we have to use a separate pointer variable to avoid memory leaks.

5 integers — 10, 20, 30, 40, 50 / 60, 70, 80, 90, 100

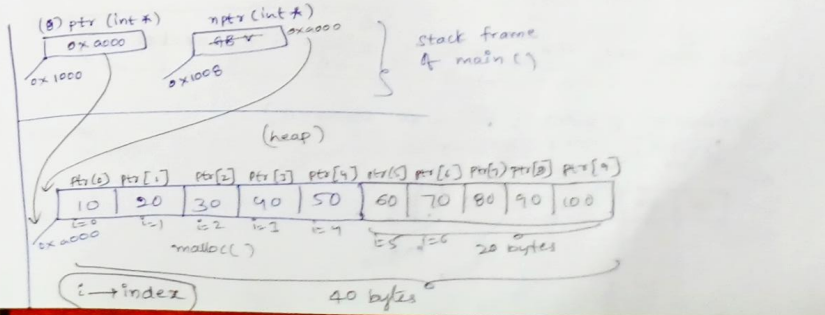
To increase the memory size using realloc ( )

```
int *ptr, *nptr;
ptr = (int *) malloc (5 * sizeof (int));   // 0x4000
if (ptr == NULL)
{
    PF("failed to allocate memory in heap segment \n");
    exit (1);
}
```

```
ptr[0] = 10;
ptr[1] = 20;
ptr[2] = 30;
ptr[3] = 40;
ptr[4] = 50;
```
(To store i/ps manually or previously)

(or)

```
for (i=0; i<5; i++)
{
    scanf("%d", &ptr[i]);
}
```
To give i/ps during run time.



0xa000

```
nptr = (int *) realloc ( ptr , 10 * sizeof(int));
                          0xa000
if (nptr == NULL)
  {
    PF(" Failed to increase the memory in heap\n");
    exit (2);
  }
```

```
Ptr [5] = 60;
Ptr [6] = 70;
Ptr [7] = 80;
Ptr [8] = 90;
Ptr [9] = 100;
```

or

```
for (i=5; i<10; i++)
  {
  scanf ("%d ", &ptr[i]);
  }
```

```
for (i=0; i<10; i++)
  {
    PF("%p - %d\n", &ptr[i] , ptr[i]);
                    &nptr[i]  nptr[i]
  }
```

```
free (ptr);   // free (nptr);
     ↓
   0xa000
```

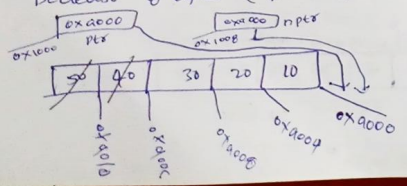
⇒ It will deallocate overall 40 bytes memory

⇒ Now, Ptr, nptr consists of same address even after deallocating that memory, then this pointers are known as dangling pointer

⇒ To deallocate the memory we can use realloc with 0 without using free ( ) & prgm termination }.

To decrease the memory size using realloc ( ).

Decrease 8 bytes (50 & 40 elements)

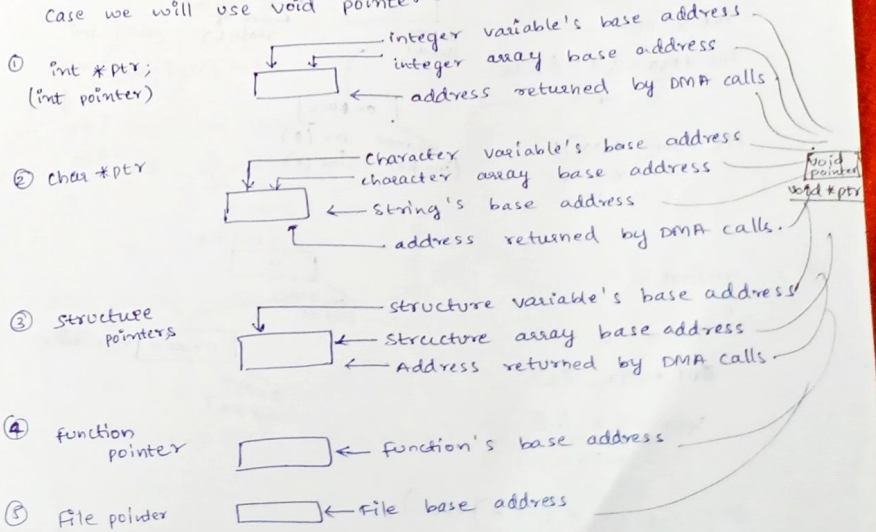



```
nptr = (int *) realloc ( ptr , 3 * sizeof (int));
                          0xa000   12 bytes
if (nptr == NULL)
  {
    printf (" Failed to decrease the memory \n");
    exit (2);
  }
for (i=0; i<3; i++)
  printf (" %p - %d \n", &nptr[i] , nptr[i]);
free (nptr);
   ↳ It will deallocate 12 bytes of memory
   Now, nptr & ptr ⇒ dangling pointers
```

* Void pointers :-

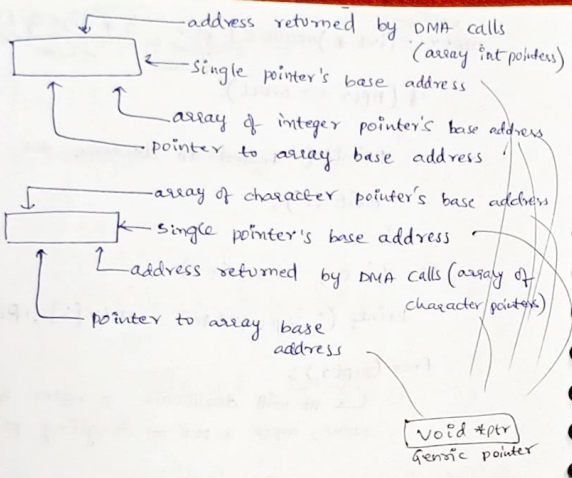
⇒ void pointer is also known as genric pointer.

⇒ If we don't know the type of data to be accessed, in this case we will use void pointer.



⑥ `int **diptr;`

double pointer



⑦ `char **dcptr`

(character double pointer)

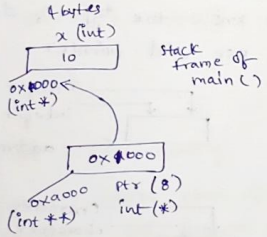
!

!

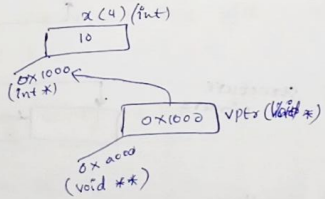
etc

⇒ To access the data by using void pointer we have to type cast the void pointer based on data

(*)
```
main()
{
  int x=10;
  int *ptr = &x; // ① & ②
  PF("%d\n", *ptr);
}
```
(&x = 0x1000; *ptr = *(0x1000) = 10)



(*)
```
main()
{
  int x=10;
  void *vptr; // ①
  vptr = &x; // ②
  PF("%d\n", *(int *)vptr);
}
```
(&x = 0x1000)



Type casted void pointer to int type to access the data.

Requesting memory for 5 elements & accessing

```
#define NM 5
main()
{
  int n, *ptr;
  ptr = (int *)malloc(NM * sizeof(int));   // 0xa000
  if (ptr==NULL)
  {
    PF("failed to allocate memory in heap\n");
    exit(1);
  }
  read(ptr, NM);
  display(ptr, NM);
  free(ptr);
}
```

```
⇒ void read(void *ptr, int n)        // int *ptr
  {
    int i=0;
    for(i=0; i<n; i++)
    {
      PF("Enter the i/p\n");
      SF("%d", &((int *)ptr)[i]);     // & ptr[i]
    }
  }
```

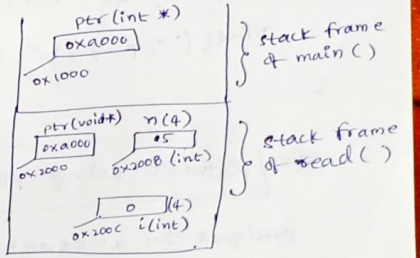
```
⇒ void display(void *ptr, int n)
  {
    int i=0;
    for(i=0; i<n; i++)
    PF("%p - %d\n", &((int*)ptr)[i], ((int*)ptr)[i]);
                                      // *((int*)ptr)+i)
  }
```

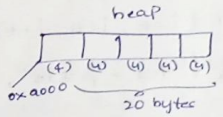
[ ] → *

return type

(*) void ⇒ That func^c will not return any value

(*) void * ⇒ That func^c will return the address which needs to be type casted. We don't know the type of data present at that address.





⇒ main()
{
unsigned int x = 0xaabbccdd;
int i;
void * ptr = &x;

// byte by byte

for (i=0; i<4; i++)

PF("%p - %x\n", &((unsigned char *)ptr)[i], ((unsigned char *)ptr)[i]);

// 2 bytes

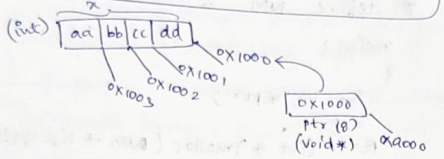
for (i=0; i<2; i++)

printf("%p - %x\n", &((unsigned short int *) ptr)[i], ((unsigned short int *) ptr)[i]);

}

(*) By using void pointer

accessing one byte & 2 bytes from 4 bytes



(*) Without using pointers

unsigned int x = 0xaabbccdd

int i;

for (i=0; i<4; i++)

// byte by byte printf("%p - %x\n", &((unsigned char *) &x)[i], ((unsigned char *) &x)[i]);

for (i=0; i<2; i++)

// 2 bytes PF("%p - %x\n", &((unsigned short int *)&x)[i], ((unsigned short int *)&x)[i]);

Assignment

→ Yesterday's prgm using void pointers

→ unsigned int x = 0x 11223344;
" " y = 0xaabbccdd;
" " z;

o/p: z = 0xaa22cc44;

Using only pointers.

① #include <stdio.h>
#include <stdlib.h>
void read_inputs (void *ptr, int n);
void display_inputs (void *ptr, int n);
void highest_digit (void *ptr, int n);
void second_highest (void *ptr, int n);
void print_binary (void *ptr, int n);
void ascending_order (void *ptr, int n);
int main()
{
unsigned int n, *ptr, opt;
PF("Enter n value (no. of elements)\n");
SF("%d", &n);
ptr = (int *) calloc (n, size of (int));
if (ptr == NULL)
{
PF("failed to allocate memory in heap\n");
exit (1);
}
while (1)
{
__fpurge(stdin);
PF("MENU \n 0. exit\n 1. read inputs\n 2. display_inputs\n 3. highest_digit\n 4. second_highest\n 5. print binary\n 6. ascending_order\n");
SF("%d", &opt);
switch (opt)
{
case 0: exit(0);
case 1: PF("Enter the n values\n");
read_inputs (ptr, n);
break;
case 2: PF("displaying the inputs\n");
display_inputs ((int *) ptr), n);
break;

```
case 3: highest_digit ((int *)ptr), n);
        break;
case 4: second_highest ((int *)ptr), n);
        break;
case 5: print_binary((int *)ptr), n);
        break;
case 6: ascending_order((int *)ptr), n);
        break;
default : PF (" Invalid option\n");
}
}
free(ptr);
}
```

① ⇒
```
void read_inputs (void *ptr, int n)
{
    int i;
    printf (" enter the i/p's \n");
    for (i=0; i<n; i++)
        scanf ("%d", &((int *)ptr)[i]);
}
```

② ⇒
```
void display_inputs ( void *ptr, int n)
{
    int i;
    for (i=0; i<n; i++)
        printf ("%p - %d\n", &((int *)ptr)[i], ((int*)ptr)[i]);
}
```

③ ⇒
```
void highest_digit ( void *ptr, int n)
{
    int i, max=0;
    for(i=0; i<n; i++)
    {
        if (max < ((int*)ptr)[i])
            max = ((int*)ptr)[i];
        PF (" highest digit is %d\n", max);
    }
```

④ ⇒
```
void second_highest (void *ptr, int n)
{
    int big=0, i=0, _2big=0;
    for (i=0; i<n; i++)
    {
        if(((int *)ptr)[i] > big)
        {
            _2big = big;
            big = ((int *)ptr)[i];
        }
        PF (" second highest is %d\n", _2big);
    }
```

else if (((int*)ptr)[i] > _2big) _2big = ((int *)ptr)[i];

⑤
```
void print_binary (void *ptr, int n)
{
    int i, j=1, rem=0, sum=0, temp;
    for (i=0; i<n; i++)
    {
        temp = ((int *)ptr)[i];
        sum = 0;
        rem = 0;
        j = 1;
        while (temp>0)
        {
            rem = temp%2;
            temp = temp/2;
            sum = sum + rem*j;
            j = j*10;
        }
        PF ("%d - %d\n", ((int*)ptr)[i], sum);
    }
}
```

⑥
```
⇒ void ascending_order (void *ptr, int n)
{
    int i, j; temp
    for (i=0; i<n ; i++)
    {
        for (j= i+1 ; j<n ; j++)
        {
            if (((int *)ptr)[i] > ((int*)ptr)[j])
            {
                temp = ((int *)ptr)[i];
                ((int *)ptr)[i] = ((int *)ptr)[j];
                ((int*)ptr)[j] = temp ;
            }
        }
        PF("%d\n", ((int*)ptr)[i]);
    }
}
```

②
```
#include<stdio.h>
int main()
{
    unsigned int x = 0x11223344;
    unsigned int y = 0xaabbccdd;
    Unsigned int z , i;
    char *ptr1 = &x;
    char *ptr2 = &y;
    Char *ptr3 = &z;
/*  ptr3[0] = ptr1[0];
    ptr3[1] = ptr2[1];
    ptr3[2] = ptr1[2];
    ptr3[3] = ptr2[3];
    printf("z = %x\n", *((int*)ptr3));  */
```
(I)

```
/*  for (i=0; i<4; i++)
    {
        if (i%2==0)
            ptr3[i] = ptr1[i];
        Else
            ptr3[i] = ptr2[i];
    }
    PF("z = %x\n", *((int*)ptr3));  */

    for (i=1; i<4 ; i= i+2)
    {
        ptr3[i] = ptr2[i];
        ptr3[i-1] = ptr1[i-1];
    }
    Printf("z = %x\n", *((int*)ptr3));
}
```
(II) (III)

01/11/2022

* Arrays :-

⇒ Arrays are used to store same type of data elements in continuous memory locations.

⇒ Arrays and pointers comes under user defined

⇒ Arrays, pointers, structures & Unions comes under Non-primitive (user defined) data types.

⇒ char, int, float, double comes under primitive data types (pre-defined)

⇒ Arrays are constructed or designed by the concept of arithmetic operations on pointers.

* Syntax of array declaration :-

<data type> <array name> [n] ;
n → no. of elements

It specifies :-
→ type of data
→ each element size
→ Range of values based on sign qualifiers