L.T.

⊛ Linux Tools ⊛

1) Intro to linux
2) Editor : Vim
3) Memory Violation tool : Electric fence & Valgrind
4) Source Code Browsing tool : ctags & Cscope.
5) Source Code Version tool : GIT.
6) Debugging tool : GDB.
7) Code Analysis tool
   - ↳ i) Static → splint & clang.
   - ↳ ii) Dynamic → g cov & lcov.
8) Compilation tool
   - ↳ (Multiple file compilation)

⇓

Makefile.

L I .

(*) Linux Internals (*).

1) Intro to Linux
   ↳ History
   ↳ Architecture & Memory layout of C-prog
   ↳ Intro to kernel

2) Kernel Services :-

1) File System	2) I/O Devices
3) Multi-Processing	4) Multi-Threading
5) Memory Mapping & Allocation	6) IPC
7) Signal Management	8) Schedular Services
9) Synchronization	10) N/W Programming
11) Device Drivers.	

Developement tools & Utilities

→ Compilation stages gcc()

→ Types of Compilation ⟹ (i) Native (ii) Cross

→ errors, bugs, debugging, failure, fault.

failure → OS Breakdown

failure → Sys h/w Breakdown

L I-2

4) System Calls in LINUX.
- ↳ Need
- ↳ Sys calls & lib fun^{n}s.

5) → Command Line Arguments
- ↳ Linker → Static & dynamic.

6) Libraries
- ↳ Need
- ↳ Types ⟶ ① Static ② Dynamic (Shared).

7) Working with files
- ↳ Sys calls of file sys.
- ↳ Linux file structure
- ↳ file related functions
- ↳ seeking a file & offset positions.

8) Linux Process
- ↳ ~~Need~~ What is Process? & its need.
- ↳ Types ⇒ Parent/Child/Zombie/Orphan/Demon.
- ↳ Alarm & timers (cover it in Signals).

LI3

9) Threads
- ↳ What is Threads? & Need.
- ↳ pthread library & API's (fun^s)
- ↳ Create thread, wait, join.
- ↳ Detachable thread & its Creation/termination
- ↳ Cancelling threads.
- ↳ Clean-up handelers.

10) Memory Management
- ↳ Need
- ↳ Memory Partitioning → Paging
  - ↳ Segmentation.
- ↳ Types of Addresses → Virtual
  - ↳ Physical
- ↳ Need of Virtual address.
- ↳ Page tables.
- ↳ Memory swapping & its need.
- ↳ Address Conversion.

  Virtual ⇌ Physical.

11) IPC
- └→ Need
- └→ Types ⟹ ① PIPE ② FIFO
  ③ Msg Q ④ Shared Memory
  ④ Semaphores ⑤ N/w Programming
- └→ Diff. betn all IPC. types

12) Synchronization
- └→ Need.
- └→ ① Semaphore ② Mutex ③ Spinlocks.
- └→ Conditional Variables.
- └→ Difference betn Semaphore &Mutex &Spinlocks.
- └→ Diff betn Binary semaphore & Mutex.

13) Schedular Processes (Management)
- └→ Need.
- └→ ① Round RObin ② Complete fair Scheduling
  (R.R) (C.F.S)
- └→ Diff betn RR & CFS

13) Signal management.

↳ What is a signal? its Need.

↳ Overview of Signal in Linux

↳ Signal handeling.

↳ Signal Library &its API's

↳ Blocking & Masking of Signals.

14) Network Programming.

↳ Need

↳ Layered Arch (OSI &TCP) & functionaliti

↳ N/w & interN/w devices.

↳ Types ⇒ (LAN | MAN | WAN)

~~E⇒~~

15) TCP/IP Stack internals.

↳ Types of N/w Addresses

↳ Diff betn (IP vs MAC)

↳ Unicast vs Broadcast vs Multicast

↳ Subnetting vs Supernetting

↳ IP & Routing Concepts.

↳ TCP vs UDP

↳ TCP dump & Raw Sockets.

↳ Ethernet

L I-G

16) Socket Programming.

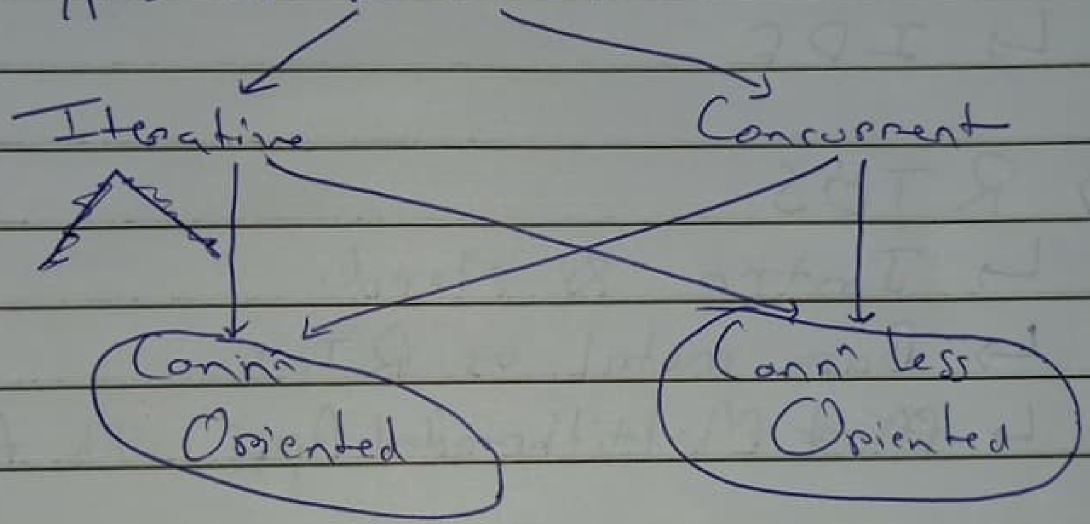
↳ Introduction & need.

↳ Socket Library & Api's (cli & serv)

↳ Conn^n & Conn^n less Orientation

↳ TCP & UDP (cli ,serv) Creation

↳ Types of servers →



⊛ Linux tools ⊛

⊛ Linux is Open source Unix like Operating System based on Linux kernel.
Released on 17 Sept 1991 by Linus Torvalds

⊛ Adv. of Linux OS.

↳ 1) Free, Open Source.
2) Portable
3) Secure
4) Scalable/Modular
5) Runs ~~24+7~~ 24*7 without Rebooting
6) Very short debugging time.
7) Suitable for programmers

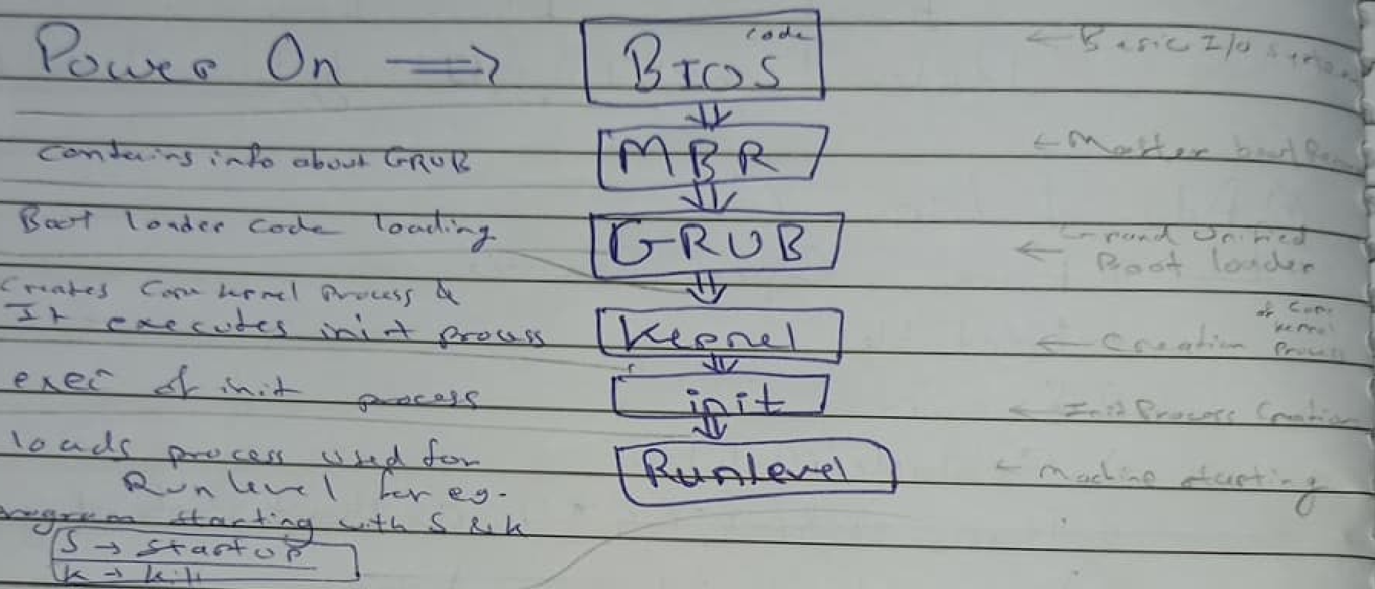
- Linux Published Under GPL { G :- G-not unix
  P :- Public
  L :- Licence

Eg. $uname -r ⇒ Kernel Version

$uname -a ⇒ Linux Version

$uname -s ⇒ OS name.

⊛ Linux Booting Sequence.



Power On ⟹ BIOS code ← Basic I/O system

contains info about GRUB — MBR ← Master boot Re

Boot loader code loading — GRUB ← Grand Unified Boot loader

creates core kernel process & It executes init process — Kernel ← Creation of core kernel process

exec^n of init process — init ← Init Process Creation

loads process used for Runlevel for eg. ~~program starting~~ with S & k — Runlevel ← machine starting

S → startup
k → kill

$ gcc --version ⟹ version of gcc.
$ ls -l output ⟹ checks "output" file created in current directory or not.

⊛ Steps of compilation (stages)

⇓

① Pre processing
② Compiling
③ Assembling
④ Linking

$$\left\{\begin{matrix}-E\\ -s\\ -c\end{matrix}\right\} \Rightarrow \left\{\begin{matrix}.i\\ .s\\ .o\end{matrix}\right\}$$

① Pre Processing → (i) Comments Removal
(.c => .i)
(ii) Macro Expansion.
(iii) File inclusion
(iv) Conditional Compilation

② Compilation ⇒ Converts english typed code into low level code (assembly level instructions)
(.i → .s)

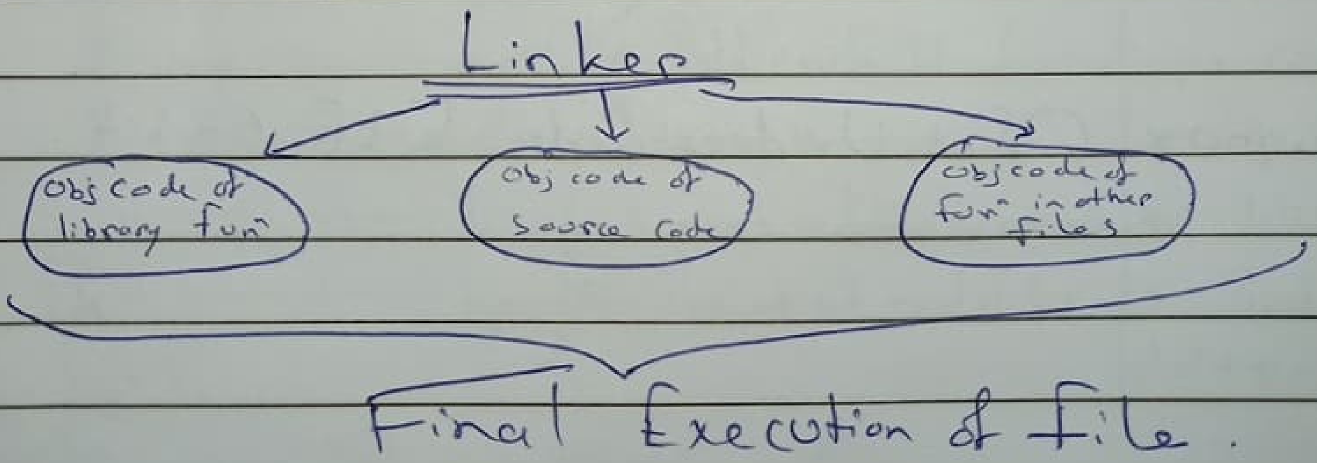
③ Assembly ~~code~~ ⇒ Converts assembly level instructions into machine Understandable code. (binary/hexadecimal) . . . (obj file).
(.s ⇒ .o)

④ Linking ⟶ Linking Library files into program.

```
$ gcc -E one.c -o one.i          (Pre Processing)

$ gce -s one.i -o one.s          (Compiling)

$ gcc -c One.s -o One.o          (Assembling)
```



Types of Linker

1) Static

```
$ gcc -static one.c -o st-out
```

Invoking Static Linker

Static Executable file

Appends entire c-library with obj code of source & creates executable.

2) Dynamic

```
$ gcc one.c -o out
```

Dynamic Executable fil

Appends only needed c-library (dynamic lib) & creates executables.

- Compiler Process data & stores in ELF (Executable Linkable format)
- Linux OS Understands ELF 64 bit.

(*) Vim

- :- Powerful text editor used in Linux & Unix Operating Sys.
- :- Two modes of using Vim

Vim

Insert mode	Command mode
:- Whatever we type reflected on screen	:- respective keys will take to cmd mode
:- act as editer	:- act as operator
:- open existing file or writes appl^n prog	

- :- Vim Operation

```
a) Vim filename        // Open vim
b) i                   // insert mode
b) esc                 // exit from insert
                          mode to cmd mode
w - save | :q - quit | :wq - save & quit
```

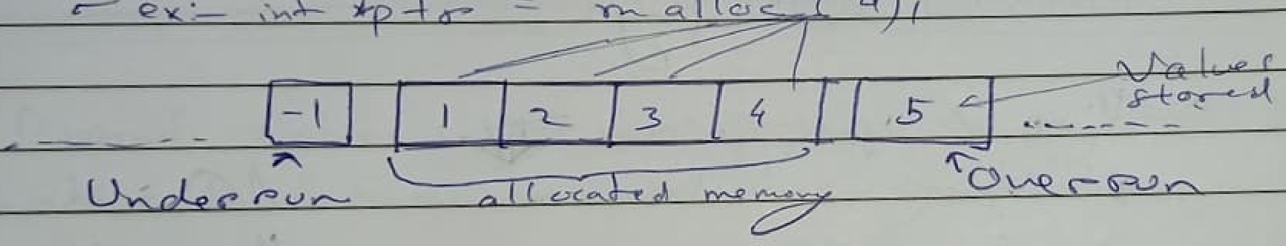
- :- Basic operations of Vim in cmd mode.

l - mv cursor to right	o - Add new ln below cursor	yy - copy ln
h - " - left	O - " - above cursor	p - paste ln
j - " - down	a - insert after cursor	x - delete ln
k - " - up	A - insert at end of line	u - undo.
		ctrl + r - redo tag

(*) Memory Violation tools

① Electric Fence ② Valgrind.

- Problem with std C library [malloc() & calloc()]
- They allocates more than user actually req.
- In linux, pg size => 4kb (4096 bytes)
- Memory divided into blocks (16/32/64/128/etc)
- if we req 17 blocks, MMU gives us 32 blocks.
  & this is more than req.
- This leads to memory violations
- ex:- int *ptr = malloc(4);



- Overrun => Writing After memory allocated
- Underrun => Writting before memory allocated.

- Gdb debugger is not effective to detect these memory violations.

1) Electroic Fence

- It is memory debugger (Memory Profiling library)
- It has own implementation of malloc() & calloc();
- In E.fence, if malloc is called, it allocates only requested memory & not more than that.
- If memory violation error occurs, efence triggers appln crash thro segmentn fault
- It is used at compilation time.

```
$ gcc -g ptr.c -o ptr -lefence          ⇒ Compilation thro efence library
$ export EF_PROTECT_BELOW=1             ⇒ to check Underrun
$ export EF_PROTECT_BELOW=0             ⇒ to check Overrun
```

- Errors & Bugs from Program point of view
  1. Dereferencing null ~~Uninitialized~~ ptr
  2. —"— Uninitilized ptr.
  3. —"— already freed ptr.
  4. writting end of array → Overrun
  5. —"— before array → Underrun

② Valgrind.

:- Stand alone Memory Debugging tool.

:- Used at runtime.

:- Identifies heap segment memory Violations

:- It is memory profiling tool & runs w/o efence library. (uses std c library malloc&calloc)

```
$ sudo snap install valgrind --classic      //installation

$ valgrind ./ptr                            //Used at runtime
```

① Segmentation fault.

:- When a prog. compiled & executed, memory segments are created for program

:- Segments ⇒ Stack // heap // bss // data // text.

:- program is supposed to use these segments.

:- If program uses memory outside these segments, this leads to Segmentation fault. & cause the application to termination

(Tip:- No of allocations must equal to no of deallocations)

(*) Source Code Browsing tool.

(1) cscope (2) ctags.

(1) Cscope

- Programming tool works with Linux & Unix OS. & Used for Source Code browsing.
- Used to find out a perticular variable, function, symbol, macro & entire project & debugging.
- It works with entire folder.

$ cscope -R ⟹ Creates cscope.out (cross Reference file)

(2) Ctags.

- Creates tags file that shows index of objects (variable name, funⁿ nam, macro nam)
- Internally uses locators. (locates where variable is used, pathnam & type of variable).

$ ctags * ⟹ Create "tags" file

$ vim tags → Opens tags file in vim.

- Advantages of ctags.

:-

1) Giving quick access ~~to~~ over several
2) Giving Complete info of function
~~3) This gives Complete info of a fun.~~
3) Gives info whether it is fun/variable.
4) tags shows global variablis information

- Redirect Operator.

:- If used, it directly sends output of an executable to file to given file only input is given on terminal.

```
$ ./out > out.txt
```

⇓

Sends output of "out" file to "out. txt" file.

✪ GIT

:- ① Source Control

- :- Provides way of tracking project files step by step. for period of time
- :- allows investigate project file.
- :- Done by saving series of snapshot
- :- It is about how project got progressed from initial stage to final stage.
- :- Most Popular Source Control ⇒ GIT.
- :- Github is appln. software works along with GIT
- :- Github allows to store repositories, provides superspace to user & allows others to add to your project

```
$ sudo apt-get install -y git
```

Git installation

✪ Git. Commands to source Control.

①
```
$ git config --global user.name "Anuj Gajbhiye19"
```
:- registering github username to local machine

②
```
$ git config --global user.email "anujgajbhiye796@gmail.com"
```
:- registering github email to local machine.

③ $git config --list

: lists username & email.

④ $ls -al

: View hidden files.

⑤ $ git clone {"URL of Repository"}

: clone repository to folder

Repository → <>code → HTTPS → copy

⑥ $git status

: check new prepared/unprepared files to repository.

⑦ $git add linux.c

: prepares "linux.c" file to add to repository.

⑧ $git commit -m "Myprog" linux.c

: file gets added to local repository in local Machi

⑨ $ git push origin main

: file gets added to local repository

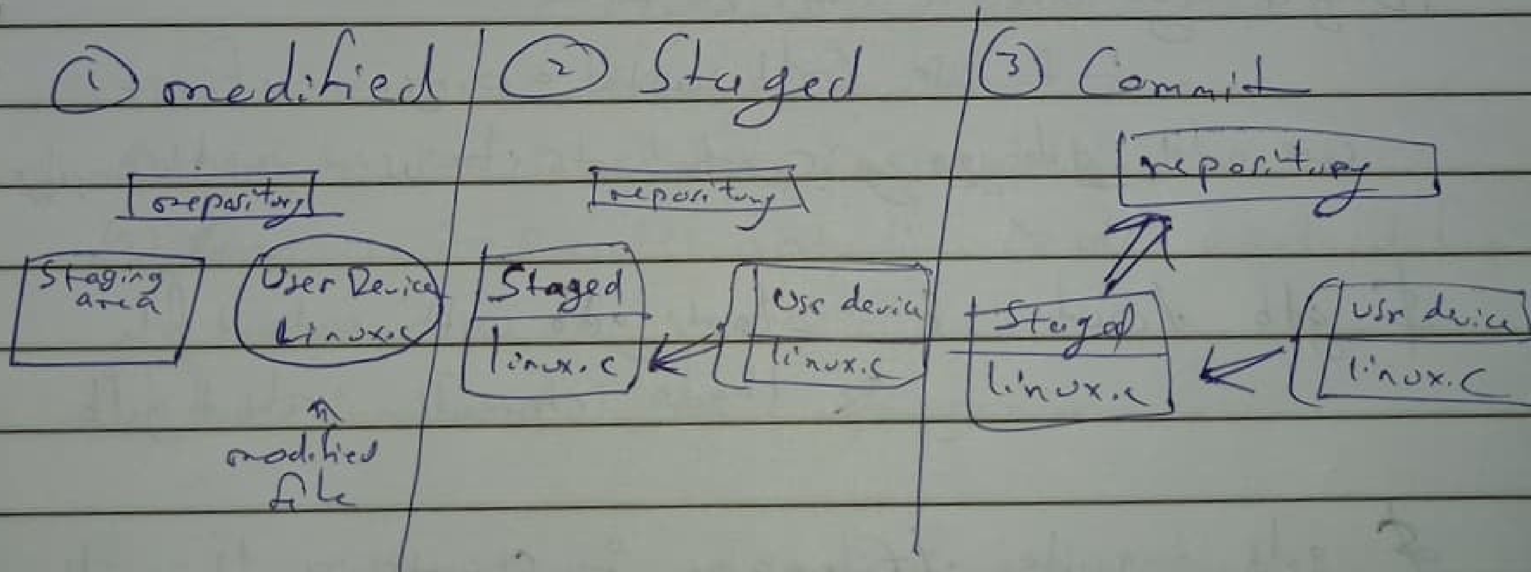
: All contents of local repository are placed into remote github server.

Note :- Github → login → acc → setting → dev. options →

↳ token → new → copy link → Use token as password

: Use this to complete push

• 3 Stages of file.



Note :- Local machine & github will maintain commit history.

☆ Debugging tool (GDB)

:- GDB is powerful debugger, free software, used in C, C++ Programming ~~Software~~ in Linux OS.

:- Program can execute within fraction of seconds & we cannot find whe~~the~~ the bug is.

:- With GDB we can control execution flow of a Program, stop & also allows to truck error.

:- It is command line tool

$ gcc -g one.c -o out

:- to use Gdb with a program, first we add debugging symbols to it usin -g while compiling.

$ gdb ./out :- Starts gdb with output file & loads information about gdb.

$ gdb --quite ./filename :- Starts gdb without showing gdb info.

- Commands in GDB.

:- ① breakpoint set ⇒ b "line no."
② run → r
③ next → n
④ step → s
⑤ list ⇒ l
⑥ print ⇒ p
⑦ information of locals ⇒ info local
⑧ quit ⇒ q

① Breakpoint :- Controls program execution.
:- Only one breakpoint is allowed.
:- Program exec^n starts & stops at breakpoint until run.

② Next :- executes next line

③ Step :- executes next line & also inserts into fun^n

④ list :- Prints whole program

⑤ print :- prints values of variables

⑥ info locals -> provides information of local variables.

- GDB is powerful tool, can jum to any loc^n using x command.

↳ p/x x → decimal to hex

↳ p/o x → dec to oct

↳ p/t x → dec to bin

↳ x/d &x → hex to bin

↳ x/o x → hex to oct

↳ x buff → jumping to "buff" address & access ascii value of present memory loc^n.

↳ x/s buff ⇒ extracting chars until we get NULL characters.

↳ set buff = "LINUX" ⇒ sets value of buff to "LINUX".

⊛ Code Analysis Tool

:- ① Static Code Analysis
   ② Dynamic Code Analysis.

① Static Code Analysis

:- Process of identifying errors & bugs at compilation time.

:- Done on some set of code by using some coding standar

:- identify loopholes & weakness of a program.

:- Analysis is done on stationary piece of code.

:- Tools (i) ~~$~~ splint (ii) clang

(i) Splint :-

```
$ sudo apt install splint

$ splint filename.c
```

:- Splint identifies → Undeclared variables
↳ undeclared functions
↳ Syntax error
↳ typedef errors

:- Used for program performance.

(ii) clang :- $ sudo apt install clang
(It is like gcc) & clang is faster

```
$ clang one.c -o out
$ clang -Wall clang.c -o out
```

↓

Shows all warning (Warning All)

⇓

used for analysis

Note :- gcc is program specific language compiler built by using c language.

:- clang → c & c++ compiler compiled by c++

(2) Dynamic Code analysis.

:- Code analysis during runtime.

(i) gcov (ii) lcov

(i) gcov :- gcc code coverage tool.

↳ open source, works along with gcc & part of gcc

↳ determines untested & unexecuted part of the code & no. of times code has run

ⓐ $ gcc -fprofile-arcs -ftest-coverage one.c -o out

∵ compile source code with this ○

& creates ~~gcda~~ .gcno file

ⓑ $ ./Out → executes code & creates ".gcda" file

ⓒ $ gcov ~~out~~.gcda → creates out.cgcov

ⓓ $ vim out.c.gcov → Open to see code analys

ⓘⓘ lcov ⇒ graphical frontend tool for gcov.

$sudo apt install lcov

repeat gcov steps ④

$ lcov -v ... shows version

$ lcov --capture --directory .--outfile 1.info

↑creates 1.info file.

$ genhtml 1.info --output-directory .-o newfile

⇓creates "newfile" folder having code analysis with index.html.

∵- performs line & fun^n coverage in order to optimise code

⊛ Compilation tool

:- (i) Make file & make

:- Used to compile multiple c files & generate one executable file

:- can mage all sourcefile & header files at once.

how to : $ create a file named "Makefile"

$ make : to execute makefile.

- Makefile :

Rules : (i) Makefile, source code, headerfiles should be in one directory

(ii) only one main function is allowed among all c files.

(iii) no repeat/duplicate functions among all c files.

: Makefile can execute shell commands (gcc)

Structure of Makefile

Variable	Margin	Target
⇓	⇓	⇓
Assigned with Values	tells where to start execuⁿ from.	thing which is going to be executed.

- target syntax :-

```
$ targetname : target dependancies
                 write command to be executed by Make
```

Note :- ~~clean~~ $ clean :

```
            rm -rf outputfile
```

```
$ make clean
```

- find :- $ find -name '*.c'

⇒ finds all files with ".c" extension in current directory.