COMMAND OVER CHAOS

THE DIRECTOR

AYMANE LAKSIMI

Command Over Chaos: Mastering the Shell to Master Yourself

the art of shell

First edition

This book was professionally typeset on Reedsy.
Find out more at reedsy.com

Contents

Preface

The command-line shell is far more than just a tool — it is a gateway to mastering the language of machines. For those of us who seek to build, understand, and create from the ground up, the shell is the first step into a world of precision, control, and power. This PDF was crafted for that purpose: to turn curiosity into competence, and hesitation into confidence.

Whether you're preparing for an intensive coding program like the 1337 Pool or simply exploring the depths of low-level programming, this guide will take you beyond memorizing commands. It's designed to help you *think* in shell, *speak* its syntax fluently, and *solve* problems like a true system craftsman.

Inside, you'll find explanations, use cases, challenges, and insights—each structured to develop not just your technical knowledge, but your way of approaching problems. The shell isn't just about typing commands; it's about learning how to see structure, patterns, and potential in the system

beneath the interface.

This is more than a PDF. It's a foundation. A weapon. A compass for those walking the path of self-taught mastery.

Welcome to the beginning.

— *The Director*

1

list directory content

how i mastered ls (list directory content)

the ‘ls’ command lets me see the contents of a directory. it’s like opening a folder in a graphical file manager. but through the raw power of terminal.

i use it to:

1. see which files are in folder
2. check file types
3. sort or filter result
4. explore any part of my system fast

Command Breakdown

ls -la

Part : Meaning
ls : List files in current directory
-l : Long format (detailed info: permissions, size, dates)
-a : Show hidden files (starting with .)

when i run ls , the shell sends a system call to read the directory structure of the current folder . Each file or folder is actually an inode on the fylesystem. ls displays info like :

1. file type (regular file, directory, link...)
2. Permissions (read , write, execute)
3. number of links
4. file owner and groupe
5. file size (in bytes)
6. last modiication time

Practice missions

always practise this

1. ls : basic list
2. ls -l : detailed view

3. ls -a : see hidden files
4. ls -la : combine both
5. ls - lh human readable sizes (1,3k instead of 1320 for example)
6. ls / list another directory
7. ls -R list recursively (see all files in all subdirectories)

Research notes

I discovered that ls is more than just listing tool it's a window into the file system .

Every time I type ls -la . i remind my self :

"i'm not just listing files i'm seeing the structure of the system."

it's my way of reading the skeleton of the OS

BONUS KNOWLEDGE :

1. files starting with . are hidden files (e.g .git

.bashrc)

2. ls -ltr —> sort by time , oldest to newest
3. ls -S —> sort by size
4. combine flags : ls -lahs (detailed , all, human, sorted)

2

search the filesystem like a hunter

syntax :

find [where_to_look] [options]

* Common examples

find . -name “*.c” # find all C files in current directory
find / -type d -name “bin” # find all directories namesd bin
find . -type f -size +1M # find all files bigger tha 1MB
find . -user unknows # files owner by you

xargs —— supercharge other commands

1. it takes output from one command and turns it

into arguments for an other

find . -name “*.txt” | xargs rm # dind all .txt files and delete them .

pro tip (safe delete):

fine . -name “*.tmp” -print0 | xargs -0 rm # -print0 & -0 handle filenames with spaces safely

chmod -R recursive permissions

chmod -R 755 myfolder
| gives rwx to owner r-x to groupe /others , foe every file and subfolder inisde

chown — change ownership

chown user:group file

example

chown unknown:1337 main.c
* you become the owner
* Groupe is now 1337

Recursive:

chown -R unknown:1337 myfolder./

make + makefile the shell's smart builder
imagine thise

You want to compile 10 .c files into 1 program
instead of writing gcc .. everytime you use make to build it only when needed

Simple 'Makefile' Example:

```
CC=gcc
CFLAGS=-Wall

all: main

main: main.o utils.o
$(CC) $(CFLAGS) -o main main.o utils.o

main.o: main.c
$(CC) $(CFLAGS) -c main.c

utils.o: utils.c
$(CC) $(CFLAGS) -c utils.c

clean:
```

```
  rm -f *.o main
```

Run with:

```
make # builds main
  make clean # deletes .o files and executable
```

It's like a smart assistant. Only rebuilds when code changes.

Advanced Shell Skills (Piscine Mastery)

FIND
- `find . -name "*.c"` → find C files
- `find / -type d -name "bin"` → find dirs
- `find . -user aymane` → by owner

XARGS
- `find . -name "*.log" | xargs rm`
- `-print0` + `-0` handles spaces

CHMOD -R / CHOWN
- `chmod -R 755 folder/`
- `chown user:group file`
- `chown -R user:group folder/`

MAKE + MAKEFILE
- ‘make‘ builds only changed code
- ‘make clean‘ deletes old builds

Example:
“‘make
main: main.o utils.o
gcc -o main main.o utils.o“‘

okey let’s understand make more

first why does makefile even exist

imagine this situation :

you’re working on a C project with many .c and .h files

let’s say:

main.c —> calls functions from utils.c and math.c

every time you change you’de need to run :

gcc -Wall -wextra -werror main.c utils.c math.c -o My_program

1. annoying
2. time-consuming.
3. if you only changed math.c why recompile all files?

make + makefile fixes this .

think of make as your personal build butler_:

1. it matches your source files
2. it understands what depends on what
3. it only compiles the files that changed
4. it compiles them the right way , every time
5. it automate and standardizes your process

so what is a makefile ?

A make file is just a file where you :

1. declare your targets (like:build, clean , etc)
2. write what commands to run
3. define dependencies between files

it looks like this :

target: dependencies
  [TAB] command to run

EXAMPLE:

main: main.c utils.c
  gcc -wall -wextra -werror main.c utils.c -o main

this is means to build main i need main.c and utils.c if either changes, ill recompile

why is this helpful in the piscine?

In 42's piscine:

1. you'll often build mutiple .c files
2. you projects like(libft, ft_printf, minishell) will require makefiles
3. you'll be evaluated on your makefile

also :

1. it shows discipline , clarity , and automation skill
2. you'll be forced to male one for every project
3. later in jobs , you'll use make , cmake, or similar

tools every day

3

The philosophy of Shell Everything is a stream

there are 3 types of data flowing in the shell:

Name	Numbers	Meaning
———	——— -	———————————————
stdin	0	what goes in (input)
stdout	1	what comes out (normal output)

Basic Commands

* echo
  Prints test (sends to stdout)

echp “Hello” # —> Hello

* cat
  Reads from a file or from stdin:

cat file.txt # shows the content of file.txt
  cat # waits for input stdin (type, then CTRL+D to finish)

* Redirection
  * > —> Redirect stdout (overwrite)

echo “heyy” > file.txt # write “heyy” into file.txt (overwrite)

* » —-> Redirect stdout (append)

echo “more” » file.txt # add “more” to file.txt (without deleting old content)

* 2> —-> Redirect stderr

ls notrealfile 2> error.txt # error goes to error.txt instead of screen

* Pipes | : Chain Commands via stdout —->

echo “hello” | cat #send output of echo to cat
  you’re connecting commands like LEGOs.
  Output of one becomes input of the next

example

ls | grep

Real world example

cat notes.txt | grep “TODO” > TODOS.TXT
WHAT happens here

1. cat notes.txt —> reads file
2. grep “TODO” — > filters lines contain TODO
3. > —> redirects to a new file called todos.txt

Bonus : combine stdout & stderr

if you want both stdout and stderr in the same file

command > all.txt 2>&1
this means :
* stdout goes to all.txt
* stderr (2) goes to same place as stdout (&1)

summary

Shell Data Flow Mastery

Streams:
- stdin (0): input
- stdout (1): normal output
- stderr (2): error output

Redirection:
- ‘>‘: stdout → overwrite
- ‘»‘: stdout → append
- ‘2>‘: stderr → redirect
- ‘2>&1‘: stderr → same as stdout

Pipes:
- ‘|‘: send stdout of one command to stdin of another

Examples:
“‘bash
echo “hi” > hi.txt
ls | grep “.c”
cat errors.txt 2> err.log

4

How linux file permissions actually work (from ground zero)

imagine linux as a multi-user kingdom . ever file is a treasure chest, and it has :

1. An owner —> the knight who created the chest (usually you)
2. A group —> guild of users allowed to access that chest
3. other —> all peasants (everyone else)

ach of these 3 groups can have **three rights**:

| Right | Symbol | Binary | Description |
| ——— | ——— | ——— | ——————————————— |
| **Read** | ‘r‘ | ‘100‘ | Can open the chest and

look |
| Write | ‘w‘ | ‘010‘ | Can edit what’s inside |
| **Exec** | ‘x‘ | ‘001‘ | Can _run_ the chest (if it’s a script) |
so each role(user/group/others) has a 3-bit flag:
like rwx —> 111 —> binary —> decimal —> 7

from bits to octal

example 1 : chmod 755 file

1. 7 = 111 —> rwx —> full access for owner
2. 5 = 101 —> r-x —> group can read & exec
3. 5 = 101 —> other can read & exec

So :

Owner : rwx = 7
group : r-x = 5
others: r-x = 5

this means

1. you can do anything
2. your team can only read and run
3. public can read and run , but not write

! Realisation : 755 is the default for scripts and programs you want others to run, but not modify.

Example 2: chmod 644 file

1. 6 = 110 —> rw- —> owner : read + write
2. 4 = 100 —> r— —-> group : read-only
3. 4 = 100 —> others : read-only

This is the default for most files you edit (e,g .txt , .conf, .html) because :

1. you edit it.
2. everyone else can view it.

ls -l Meaning

-rwxr-xr— 1 aymane devs 1234 may 31 script.sh

Symbol	Meaning
————	————————————————————
‘-‘	regular file (can be ‘d‘ for directory)
‘rwx‘	owner permissions
‘r-x‘	group permissions

‘r—‘	others permissions
‘aymane‘	the owner
‘devs‘	the group

Every file carries its own “mini access policy” right inside it . that’s how linux remains secure by design.

who stets these automatically ?

1. when you create a file , you become the owner
2. your group is usually the one tied to your user (id -gn)
3. linux applies default permissions using umask

for examples

umask 022
Files : 666 - 022 = 644
dirs : 777 - 022 = 755

so most files are born with :

1. rw-r—r— —> 644

And most folders/scripts:

1. rwxr-xr-x —> 755

Realization

visualize:

USER	GROUP	OTHERS	
—	— -	— — —	— — — — — — — — -
rwx	r-x	r-x	755 (script)
rw-	r—	r—	644 (doc)
rwx	- - -	- - -	700 (private key)
you think in permissions like a builder of unix .

train :

Binary	octal	Symbolic
— — —	— — -	— — — —
111	7	rwx
110	6	rw-
101	5	r-x
100	4	r—
010	2	-w-
000	0	- - -

File Permission Breakdown

Example: -rwxr-xr—

- **Owner** (rwx = 7): read + write + execute
- **Group** (r-x = 5): read + execute
- **Others** (r— = 4): read only

Total = 754 or 755 (if others had +x)

How I Think Now:
I don't memorize '755'.
I visualize:

USER: rwx
GROUP: r-x
OTHERS: r—
→ Meaning: I code, others run, nobody changes.

Binary Brain Model of Linux Permissions

permission = 1 bit:
- 'r' = read = 1
- 'w' = write = 1
- 'x' = execute = 1
- '-' = off = 0

Examples:
- 'rwx' = 111 = 7
- 'rw-' = 110 = 6
- 'r-x' = 101 = 5
- 'r—' = 100 = 4
- '—-' = 000 = 0

Realization:

Every time I see ‘chmod 755‘, I think:
- Owner: ‘rwx‘ = 7
- Group: ‘r-x‘ = 5
- Others: ‘r-x‘ = 5

I read permissions **like I invented them.**

5

How i gained full controle over file permissions with chmod & chown

in a unix-like system . everything is a file and every file has permissions that controle who can read , write, or execute it .

As a system-level user, i use :

1. chmod —> to change permissions
2. —> to change ownership

If i want complete comtrole over who does what with my files , thes are the keys .

Command breakdown

chmod 755 script.sh

chmod : change file mode (permissions)
755 : octal code : owner = 7 , group = 5 , other 5.
script.sh : target file (could be any file).

chown root:admin file.txt
chown : chhange file ownership
root:admin : new owner:group
file.txt : target file

Under The Hood : How linux Handles Permissions
every file has

1. Owner
2. group
3. permissions for : Owner / Groupe / Others

Practice missions

1. create a file

touch hello.sh && echo ‘hello world’ > hello.sh
2. Remove execute permissions :

chmod -x hello.sh
./hello.sh
3. add execute permission:

chmod +x hello.sh
./hello.sh # _
4. change ownership (need sudo)

sudo chown $USER:$(id -gn) hello.sh

My research Notes

i realized that Linux treat every file like a locked box , and i'm the locksmith.

Using chmod , i can grant or restrict access like a king:

1. want to keep a file private ? chmod 700
2. want it public-readable but safe? chnod 644
3. want a script to be executable ? chmod +x

and with chown , i decide who owns the box.

most beginners fear permissions but i see them as

power tools.

Bonus Flags & Wisdom

1. chmod -R 755 folder/ —> change permissions recursively
2. chown -R user:groupe dir/ —> change ownership of everything inside
3. ls -l —> always check permissions in symbolic form (-rwxr-xr-x)
4. umask —> default permissions mask on file creation

traps to avoid

1. if you forget +x on scripts —> permission denied
2. using chmod 777 blindly = security risk
3. if you're not root —> chown will fail

6

Environment Variables

what is an environment variable?

it's like a named value stored in your shell's memory, it affects how programs run

Example:

```
echo $HOME # shows your home directory
  echo $USER # shows your current user
```

you can create your own :

```
export NAME="Aymane"
  echo $NAME # output : Aymane
```

they're like global variables for your terminal . Programs like gcc , vim, make all use them

the $path variable

1. what is PATH?

it's a special environment variable that tells your shell where to look for commands.

echo $path

outpur might look like :

/usr/local/sbin:/usr/local/bin:/usr/bin:/bin:/usr/sbin:/sbin

so when you type ls, the shell checks these folders in order to find the ls program

2. you can add to it

export PATH=$PATH:/home/unknown/my_tools

now lets talk about bashrc or .zshrc

1. what are they
   they are startup config files for your shell:

   1. .bashrc —-> for Bash shell
   2. zshrc —> for zsh shell (often used in 42)

every time you open a new terminal or shell, it runs the commands in those file.

you can add aliases

alias gs='git status'

set environment variable permanently:

export PATH=$PATH:/YOUR/COSTUM/FOLDER

crate functions
after editing :

source ~/.zshrc #or source ~/. bashrc

Shell sessions
1. What happens when you open a terminal
* it starts a shell session
* it loads .bashrc or zshrc
* it reads your environment
* you're ready to run commands

A shell session ends when you :

exit

summary :

Shell Concepts — Internalized

1. Environment Variables
- KEY = VALUE stored globally
- ‘echo $VAR‘ to read it
- ‘export VAR=...‘ to create or update
- Used by apps, compilers, shell

2. PATH
- List of folders searched to run commands
- ‘echo $PATH‘ to view it
- Add to it → ‘export PATH=$PATH:/my/path‘

3. .zshrc / .bashrc
- Run every time shell starts
- Add aliases, PATH, env vars
- ‘source ~/.zshrc‘ to reload manually

4. Shell Session
- Starts when you open a terminal
- Loads your shell configs
- Ends with ‘exit‘ or closing the window