page 0-100

🪖 Intel Report: Linux Operations – Pages 0 to 100

📂 Recon on `ls` and Inode Targeting:

- `ls` → Lists files/directories in the current location.
- `ls -li` → Enhanced listing with:
  - `-l`: Detailed long format (permissions, owner, size, etc.).
  - `-i`: Displays **inode number** (unique identifier for each file in the filesystem).

🔗 Link Strategy – File Persistence Analysis:

- `ln` (Hard Link):
  - Creates a direct pointer to the file data block.
  - Survives if the original file is deleted.
  - Independent storage reference.
- `ln -s` (Symbolic Link):
  - Creates a shortcut to the original file path.
  - **Breaks** if the original file is deleted.
  - Path-based, not data-based.

🔄 I/O Control – Redirection Tactics:

- `>` → Redirect STDOUT to a file (overwrites).
- `>>` → Append STDOUT to file.
- `<` → Feed file as STDIN to a command.
- `2>` → Redirect STDERR.
- `&>` → Redirect both STDOUT and STDERR.
- `command > file 2>&1` → Merges both STDOUT and STDERR into a single file.

🔁 Pipeline Protocol – Stream Chaining:

- `|` → Forwards output of one command as input to the next.

Example:

```
ls /bin /usr/bin | sort | less
```

Mission: Enumerate binaries, sort alphabetically, pipe to pager.

🔧 Utility Command Arsenal:

- `uniq` : Eliminates duplicate lines.
  - `-d` : Displays only repeated lines.
- `sort` : Sorts input lines.
- `wc` : Counts lines, words, characters.
- `tee` : Outputs to terminal **and** saves to file.
- `xargs` : Constructs and executes commands from piped input.

🔗 Command Chain Discipline:

- `;` → Execute commands sequentially (no condition).
- `&&` → Execute second only if first **succeeds**.
- `||` → Execute second only if first **fails**.

🧬 Wildcard & Expansion Mechanics:

- `*`, `?`, `[a-z]` : Glob patterns for filename expansion.
- `.*` : Matches all hidden files.
- `.[!.]*` : Matches hidden files **excluding** `.` and `..`.
- `{}` : Brace expansion – generates patterns.

```
echo Front-{A,B,C}-Back # Output: Front-A-Back Front-B-Back Front-C-Back
```

🎯 Subshell Execution Intel:

- `` `command` `` and `$(command)` :

- Both execute subshells and substitute the result.
- `$(...)` is **preferred** for readability and nesting.

🔪 Directory Navigation Awareness:

- `.` → Current directory.
- `..` → Parent directory.

Used for navigation, scripting, and path-relative operations.

🕶️ Intel Addendum – Hidden File Ops & Stream Sanitization

🪞 Hidden Asset Detection – Dotfile Recon

- `ls -a` → Lists **all** files including dotfiles (`.` and `..`).
- `ls -d .[^.]*` → Isolates **only hidden files**, excluding `.` and `..`.

> **Pattern logic**:
>
> - `.` → Starts with a dot
> - `[^.]` → Followed by **any character except a dot**
> - `*` → Followed by zero or more of any character

Mission: Clean visibility on stealth files (config, secrets, cloaked assets).

📑 Data Stream Filtering – `sort` & `uniq` Protocol

- `sort` : Arranges input lines in ascending order (alphabetical or numeric).
  - `-r` : Reverse order.
  - `-n` : Numerical sort.
  - `-u` : Unique lines only.
- `uniq` : Removes **adjacent** duplicate lines.
  - `-d` : Shows only duplicated lines.
  - `-u` : Shows only unique (non-repeating) lines.
  - `-c` : Prefixes line count.

🔁 **Tactical Combo**:

bash

```
sort intel.log | uniq -d
```

Objective: Identify repeated intelligence logs post-sorting (non-contiguous duplicates).

🧠 **Operator Memo**: These tools are fundamental for sanitizing large output logs, config audits, and anomaly detection routines.

⚠️ **Status: Operator has completed a critical range (pages 0–100) of Linux field training. Progress confirms deep comprehension of core shell navigation, redirection, and command logic.**
Next move: **Proceed to GitHub and JavaScript protocol training.**

page 100-130

Absolutely. Here's your notes rewritten into a **concise, powerful, intelligence-agency style summary** — straight, sharp, and field-ready.

🔍 Linux Intelligence Dossier: Command Line Ops & Permissions

📡 Command Output: `echo` Variants & Escape Sequences

```
echo hello world # Plaintext echo "hello world" # Quoted echo -r "hello world \t ..." # Invalid flag (-r), ignored in Bash echo 'hello world $((2+2)) \n {a..z}' # Single quotes = raw literal echo "lol" $\a' "go to gym!!" # Odd syntax, \a = bell char (sound)
```

- `\t` → horizontal tab
- `\a` → alert (beep)
- `\b` → backspace
- `\n` → new line

🧠 *Single quotes prevent variable expansion. Double quotes allow it.*

🔧 Keyboard Navigation Commands (Command Line Intel)

Key Combo	Effect
`Ctrl + A`	Go to line **start**
`Ctrl + E`	Go to line **end**
`Ctrl + F` / `Ctrl + B`	Move **forward/back** one char
`Alt + F` / `Alt + B`	Move **forward/back** one word
`Ctrl + L`	**Clear screen** (same as `clear`)

📜 Tab Completion & History Recall

Sequence	Action
`Alt + ?` / `TAB` x2	List **all completions**
`Alt + *`	Insert **all matches**
`!!`	Run **last command**
`!number`	Run command from **history by index**
`!string`	Run last command starting with `string`
`!?string`	Run last command **containing** `string`

🛡️ File Permissions

Basic:

- `r` = read
- `w` = write
- `x` = execute

File Type Indicators (first char in `ls -l` output):

- `-` = regular file
- `d` = directory
- `l` = symbolic link
- `c` = character device
- `b` = block device

Example:

```
-rwxr-xr-- 1 root staff ... ↑ ↑ ↑ | | └ others | └─── group └──────── owner
```

🎯 Special Permissions (Setuid, Setgid, Sticky Bit)

🔥 Setuid (u+s)

- Applies to: **files**
- When set, file executes with the **owner's privileges**.
- `chmod 4755 file`
- Appears as: `-rwsr-xr-x`

- Used for: `passwd`, `sudo`, etc.

🛡️ Setgid (g+s)

- Applies to: **files & directories**
- On **files**: executes with **group privileges**.
- On **directories**: newly created files **inherit group**.
- `chmod 2755 dir`
- Appears as: `drwxr-sr-x`

🩹 Sticky Bit (o+t)

- Applies to: **directories**
- Only file owners (or root) can delete/rename files inside.
- Common in `/tmp`
- `chmod +t dir`
- Appears as: `drwxrwxrwt`

🔐 Numeric Special Permissions

Format: `chmod [special][owner][group][others]`

Octal	Meaning	Example
`4`	setuid	`chmod 4755`
`2`	setgid	`chmod 2755`
`1`	sticky bit	`chmod 1755`

Example:

```
chmod 7777 file # Gives rwxrwxrwx + all special bits (suid, sgid, sticky)
```

`chmod 3777` = sgid + full permissions
`chmod 1777` = sticky + full permissions

🧑‍💻 User Switching: sudo vs su

- `sudo` → Run a single command as root

- `su` → Switch user (default: root), full login not always included
- `su -` or `su -l` → Full root login shell (`$HOME`, `$PATH` updated)
- `sudo -i` → Interactive shell (depends on shell behavior)

`/etc/sudoers` → File that controls who can use `sudo`.
Edit via `visudo` to prevent syntax errors that could lock out root.

🎯 Mission Recap

You're now operating with:

- Echo precision
- Keyboard-level mobility
- Shell history recall tactics
- Mastery over permission bits (standard + special)
- User escalation via `sudo`, `su`, and shell flags

> 🕵️ *You're not just learning Linux — you're infiltrating it.*
> Keep digging. Deeper control means deeper power.

🗒️ Ownership Manipulation — Classified Briefing

🔧 `chown` – Change Owner

Syntax:

```
chown [NEW_OWNER] [FILE] chown [OWNER]:[GROUP] [FILE]
```

Purpose:

- Reassign **user ownership** of a file/directory.
- Can optionally assign **group** at the same time.

Examples:

```
chown bill file.txt # file.txt becomes owned by user 'bill' chown bill:music song.mp3 # owner = bill, group = music chown :devs script.sh # only change group to 'devs'
```

📌 **Requires sudo** — only root can change ownership.

🧪 Real Ops Example:

```
sudo chown :music /usr/local/share/Music
```

🟢 This changes the group of `Music` dir to `music`, enabling group collaboration.

🔧 `chgrp` – Change Group

Syntax:

```
chgrp [GROUP] [FILE]
```

Purpose:

- Change **group ownership only** — quicker when you don't need to change the user.

Example:

```
chgrp project confidential.doc
```

🟡 `chgrp` is basically a **subset** of `chown`, used just for group switching.

🔐 Why Ownership Matters:

- Linux security is based on **user:group:other** model.
- If ownership isn't right, **permissions are useless** — you either get "Permission denied" or unwanted access leaks.

💡 **Intel Tip**:
Use `ls -l` to verify ownership at any time.

```
ls -l file.txt # output: -rw-r--r-- 1 bill devs 1234 Jan 01 00:00 file.txt # ↑ ↑ # owner group
```

> 🧠 "Control ownership, control access. That's the rule of engagement."

page 130 -150

🛰️ [CLASSIFIED DOSSIER] — Terminal Operations: Process & Shell Environment

> **Clearance Level:** TOP SECRET
> **Issued To:** Operative Cybryon
> **Purpose:** Tactical mastery of process management, signal control, and shell-based infiltration protocols

---

🎯 SECTION 1: Process Surveillance & Control

📍 `ps` – Process Scan

> **Purpose:** Recon current field agents (processes)

`ps aux # BSD-style, shows all processes ps -ef # GNU-style, same intel different flavor`

Key Intel Fields:

Header	Meaning
`USER`	Process owner (agent in charge)
`PID`	Process ID (target tag)
`%CPU`	CPU usage (%) – how aggressive
`%MEM`	RAM usage (%) – resource footprint
`VSZ`	Virtual memory (KB) – total allocated
`RSS`	Resident memory (KB) – actual footprint
`STAT`	Status code – see below
`START`	When the agent activated
`COMMAND`	The mission (program) being executed

Process Status Codes:

Code	Intel
`R`	Running – active on mission
`S`	Sleeping – waiting for signal
`D`	Deep sleep – I/O wait, uninterruptible
`T`	Terminated/stopped – suspended
`Z`	Zombie – dead, but not cleaned
`Ss`	Sleeping + session leader
`Sl`	Sleeping + multithreaded
`R+`	Active + foreground ops

📍 `top` – Live Recon Stream

> **Purpose:** Real-time surveillance of CPU battlefield

Notables:

- **Load Average:** Number of ops waiting — 1, 5, 15 min window
- **CPU stats:**
  - `%us` : user-mode agents
  - `%sy` : kernel agents
  - `%ni` : nice (low-priority) ops
  - `%id` : CPU idle
  - `%wa` : waiting on I/O

Memory Units:

- **Mem:** Physical RAM stats
- **Swap:** Virtual memory usage

📍 `jobs` , `fg` , `bg` , `&` , `%` – Mission Control (Local Ops)

Command	Function
`&`	Launch op in background

Command	Function
`ctrl+Z`	Suspend mission (send `TSTP`)
`jobs`	See active/sleeping background ops
`fg %1`	Bring background mission #1 to front
`bg %1`	Resume #1 in background

📍 `kill` – Signal Dispatch Protocol

> **Purpose:** Transmit SIGINT or lethal action to rogue agents

Example:

```
kill -9 1234 # KILL with extreme prejudice
```

Signal Directory:

No.	Signal	Action
1	`HUP`	Hangup — reinitialize or drop
2	`INT`	Interrupt — polite stop (`Ctrl+C`)
3	`QUIT`	Abort with core dump
9	`KILL`	Immediate termination – NO RETURN
15	`TERM`	Soft kill – graceful exit
18	`CONT`	Continue mission
19	`STOP`	Hard pause (non-ignorable)
20	`TSTP`	Terminal pause (like Ctrl+Z)
11	`SEGV`	Segfault – illegal memory access
28	`WINCH`	Terminal window resize

> To view all signals: `kill -l`

🧠 SECTION 2: Shell Environment Ops

🔎 Shell Variables vs. Environment Variables

Type	Scope	Exportable?
Shell Variable	Local only	❌ No
Environment Variable	Inherited by child processes	✅ Yes

```
name=Cyber # Shell variable export name # Now it's env variable
```

🧬 `printenv`, `set`, `export`, `alias` – Tools of the Trade

Command	Mission Purpose
`printenv`	Show exported (env) variables
`set`	Show all shell and env vars, + functions
`export VAR=value`	Promote shell var to global
`alias`	Create command shorthand
`unalias`	Remove shorthand

🧩 Sample Environment Variables

Var	Description
`PATH`	Directories for locating executables
`HOME`	Agent's HQ (home folder)
`USER`	Codename
`SHELL`	Type of shell
`EDITOR`	Default text editor
`TERM`	Terminal type
`DISPLAY`	X11 session display
`LANG`	Language + character set
`PS1`	Prompt customization
`PWD`	Present working directory
`OLDPWD`	Last visited directory
`TZ`	Timezone config

🏁 SECTION 3: Shell Session Initialization

🔐 Login Shell Session (auth via tty/SSH)

File	Purpose
`/etc/profile`	Global startup ops (all agents)
`~/.bash_profile`	Personal ops file
`~/.bash_login`	Backup if above absent
`~/.profile`	Fallback for Debian-based ops

🔓 Non-login Shell Session (GUI Terminal ops)

File	Purpose
`/etc/bash.bashrc`	Global config for terminal
`~/.bashrc`	User terminal behavior

> 🔁 Many agents embed `.bashrc` inside `.bash_profile` to unify the setup

```
# Inside ~/.bash_profile [[ -f ~/.bashrc ]] && source ~/.bashrc
```

⚠️ SECTION 4: System Shutdown Protocols

Command	Action
`shutdown -h +12`	Halt system in 12 minutes
`shutdown -r +12`	Reboot in 12 minutes
`shutdown -c`	Cancel scheduled shutdown
`halt`	Immediate stop (no power off)
`poweroff`	Shutdown + power off
`reboot`	Restart the system

🧩 SECTION 5: System Monitoring Utilities

Command	Purpose
`pstree`	Visualize process parent/child structure
`vmstat 5`	Resource usage every 5s
`xload`	Graphical load monitor
`tload`	Terminal graph of system load

🎓 Mission Debrief

> This briefing covers tactical control of Linux systems from a command-line perspective. Mastery here provides **core skills** for:

- Cybersecurity recon and exploit chaining
- Privilege escalation and persistence
- System forensics and live monitoring
- Penetration testing and Red Team ops

🗒️ OPERATION: ENV_VARS — INTEL BRIEF

> **Classification:** Internal Ops
> **Agent:** Cybryon
> **Mission Segment:** Environmental Variables — Behavioral and Tactical Layer
> **Status:** Phase I Complete

🔍 PRIMARY FINDINGS

1. **Shell vs Environment Variables**
   - *Shell Variables* are **local** to the shell. They exist within the current session and are not inherited by child processes.
   - *Environment Variables* are **exported** to subprocesses. Used by applications, tools, and scripts for operational behavior.
2. **PS1: The Silent Operator**
   - Although **not an environment variable**, `PS1` behaves like one in effect.
   - It governs the **visual structure** of the terminal prompt.

- Not visible through `printenv` — it's a **non-exported shell variable**, only available within the shell unless explicitly exported (not recommended).

3. **Storage & Access Points**
   - **Global level**: `/etc/environment`, `/etc/profile`, `/etc/bash.bashrc`
   - **User level**: `~/.profile`, `~/.bash_profile`, `~/.bashrc`
   - Behavior changes based on shell type:
     - `bash_profile`: **login shells**
     - `bashrc`: **interactive non-login shells** (e.g., terminal emulator)
4. **PATH Manipulation**
   - The `$PATH` variable defines **execution scope**.
   - Adding a directory in the **beginning** of `$PATH` gives it **execution priority** — useful for overrides or controlled redirection.
5. **Persistence Protocol**
   - Any temporary change to variables (e.g., via `export`) is **volatile** and lost upon session termination.
   - For **persistent deployment**, modifications must be written to `~/.bashrc` or `~/.profile` depending on use case.

🛠️ FIELD TOOLS

Command	Usage
`echo $VAR`	Read value
`export VAR=value`	Promote to environment level
`printenv` / `env`	List environment variables
`declare -p VAR`	Inspect variable type and export status
`source ~/.bashrc`	Reload user config

⚠️ INTEL NOTE

> PS1, PATH, LANG, EDITOR — these are not just variables; they are **behavioral controls**. Master them, and you master the shell's behavior under every condition.

🧠 **Next Objective**: Advance deeper into shell behavior — aliases, functions, scripting control, and command substitution.

🛰️ Log your traces. Validate your understanding through real-time changes. Remember: mastery lies not in memorization, but in command-line execution under pressure.

🧳 **Next Mission Objective:**

page 150-180

Stay sharp, Operative. The terminal is your battlefield.

page 150-180

🧠 INTELLIGENCE FIELD REPORT: Section — vi Basics & PS1 Mechanics

---

🎯 OPERATION: Text Editor Survival — `vi` Essentials

- **vi is everywhere**. This is not optional. POSIX mandates its presence. If you're stuck in the field (non-GUI environments), `vi` is your escape tool. Learn it, or be left behind.
- In case you're using `vim`, it may behave in **compatibility mode** (like traditional vi). To unlock its full potential: bash Copy code `echo "set nocp" >> ~/.vimrc`
- **Insertion/Command Modes:**
  - `i` – Enter insert mode at cursor.
  - `a` / `A` – Insert after cursor / end of line.
  - `0` – Move to beginning of line.
  - `o` / `O` – Open a new line below/above.
  - `u` – Undo (note: real `vi` supports only one level).
  - `G` – Go to the last line.
  - `dd` – Cut/delete entire line.
  - `x` – Delete character under cursor.
  - `p` – Paste after cursor.
  - `yy` – Copy entire line.

🔍 Search and Replace Recon

- **Search:**
  - `f<char>` – Find the next occurrence of character in the current line.
  - `/word` – Search forward for word.
- **Replace:** vim Copy code `:%s/old/new/gc`
  - `:` – Command mode.
  - `%` – Apply to all lines.
  - `s` – Substitute.
  - `g` – Replace all matches per line.
  - `c` – Confirm before replacing.

🧭 Multi-File Navigation

- Open multiple files: bash Copy code `vi file1 file2`
- Navigation:
  - `:buffers` – List all files in session.
  - `:bn` / `:bp` – Next / previous file.
  - `:buffer N` – Switch to file N.
  - `:e filename` – Open new file.
  - `:r filename` – Read another file's content into current buffer.

💾 Saving Mission Progress

- `:w filename` – Write/save.
- `ZZ` – Save and quit (classic escape).

🧬 OPERATION: Customizing the Prompt — `PS1` Protocol

- **Purpose:** `PS1` defines the shell prompt appearance. It's not an environment variable (not inherited by child processes), but a shell variable that controls UX visuals in interactive terminals.
- **Sample Format:** bash Copy code `PS1="< \u \T \[\a\] >"` Output: Copy code `< cybryon 14:20 > + bell sound` 🔔
- **Common Escape Sequences:**

	Code
	`\u`
	`\h` / `\H`
	`\w` / `\W`
	`\t` / `\T` / `\@`
	`\d`
	`\!` / `\#`
	`\$`
	`\a`
	`\[` `\]`

  ⚠️ *Anything inside `\[` and `\]` doesn't display, but still affects behavior.*
- **Why PS1 isn't exported:**
  It's shell-local. Child processes don't need to know how your terminal *looks*, only how it *functions*. Exporting it would pollute environment unnecessarily.

📎 Final Note:

These features — from `vi` to `PS1` — are not just quality-of-life upgrades. They are **tactical tools** in low-level Unix-based environments. Mastering them turns a shell into a weaponized workspace

---

"next mission page 180-210 stay sharp agent !"

page 180-213

🧠 INTELLIGENCE FIELD REPORT: Terminal Control & PS1 Engineering

---

🎯 OPERATION: Terminal Control Systems

- **terminfo** = database of terminal control codes.
  → Example: `clear` asks terminfo for the correct code for your terminal and runs it.
- Some commands (like `echo -e "\033[0;31m"`) bypass terminfo and write escape codes directly → faster, but less portable.

⚙️ OPERATION: `tput` — The Portable Tool

- `tput` reads from **terminfo** and prints the correct escape code.
- **Pros:** portable (works on different terminals).
- **Cons:** slower (needs DB + system calls).
- Speed test:
  - `echo -e` → ~0.012s (1000 times).
  - `tput setaf 4` → ~1.0s (1000 times).

🎨 OPERATION: ANSI Escape Codes

- General form: `\033[<attribute>;<color>m`
- Examples:
  - `\033[0;31m` → red.
  - `\033[1;34m` → bright blue.
- **Text colors (30–37):** black, red, green, yellow, blue, purple, cyan, light gray.
- **Background colors (40–47):** same colors, but as background.
- **Extra attributes:**
  - `1` = bold (not always supported for background).
  - `5` = blink.

- `7` = reverse video.

🧭 OPERATION: PS1 Engineering

❗ Problem

- If you put raw escape codes in `PS1`, Bash miscalculates the length.
- Result: cursor jumps or breaks when typing long commands.

✅ Fix

- Wrap escape codes inside `\[...\]` so Bash knows they are invisible.

Correct example:

```
PS1="\[\033[0;31m\]<\u@\h \W>\$\[\033[0m\] "
```

🖼️ ADVANCED TACTIC: Status Bar Prompt

```
export PS1="\[\033[s\033[H\033[0;41m\033[K\033[1;33m\t\033[0m\033[u\]<\u@\h \W>\$ "
```

Breakdown

Part	Technical meaning	Result	Note
`\[...\]`	Mark sequence as invisible	Prevents line break issues	Mandatory with escape codes
`\033[s`	Save cursor position	Stores starting point	Used later
`\033[H`	Move cursor to top-left	Cursor goes to 0,0	Safer than `0;0H`
`\033[0;41m`	Red background	Creates red bar	
`\033[K`	Erase to end of line	Fills whole line	
`\033[1;33m`	Bold + yellow text	Yellow on red	
`\t`	Bash time (HH:MM:SS)	Shows time	Auto updates
`\033[0m`	Reset attributes	Back to normal	Prevent color leak
`\033[u`	Restore cursor pos	Return to start	Prompt prints normally

Part	Technical meaning	Result	Note
`<\u@\h \W>\$`	Real prompt	`<user@host dir>$`	Standard prompt

🧪 FIELD TRAINING

1. Run:

   ```
   echo -e "\033[1;34m Blue \033[0m Normal"
   ```

   → See if the text turns blue.
2. Compare speed:

   ```
   time for i in {1..1000}; do echo -e "\033[31m" >/dev/null; done time for i in {1..1000}; do tput setaf 1 >/dev/null; done
   ```

   → Notice echo is much faster.
3. Create your own PS1 with different colors (e.g., green text on blue background).

📎 Final Note

- **terminfo** = the official database.
- **tput** = portable, but slower.
- **echo + ANSI** = faster, but less portable.
- **PS1** must use `\[...\]` for correct behavior.

🧠 INTELLIGENCE FIELD REPORT: Repository & Package Operations — Debian vs Red Hat

---

🎯 OPERATION: Understanding the Supply Chain

- **Repository** = a remote warehouse of software packages.
  → Example: `deb.debian.org` for Debian, `mirror.centos.org` for Red Hat.
- **Package** = the actual software unit (binary + metadata).

- **Dependencies** = other packages required for a program to work.
  → Example: installing `nmap` may pull in `libpcap` automatically.
- **Key distinction:**
  - Repository = *where* software comes from.
  - Package = *what* you install.

⚙️ OPERATION: Debian-Based Systems (APT/Dpkg)

- **Update repository list:** `sudo apt update`
- **Upgrade installed packages:** `sudo apt upgrade`
- **Full upgrade (handles deps/removals):** `sudo apt full-upgrade`
- **Install software:** `sudo apt install package_name`
- **Remove software (keep config):** `sudo apt remove package_name`
- **Purge software (remove configs too):** `sudo apt purge package_name`
- **Dependency check:** `apt depends package_name`
- **Show reverse dependencies (who needs it):** `apt rdepends package_name`

🔧 OPERATION: Red Hat-Based Systems (DNF/YUM/RPM)

- **Update repo metadata:** `sudo dnf check-update`
- **Upgrade installed packages:** `sudo dnf upgrade`
- **Install software:** `sudo dnf install package_name`
- **Remove software:** `sudo dnf remove package_name`
- **Check dependencies:** `dnf deplist package_name`
- **Show package info:** `rpm -qi package_name`
- **List installed files for a package:** `rpm -ql package_name`

🧭 Comparison: Debian vs Red Hat

Task	Debian (APT/Dpkg)	Red Hat (DNF/RPM)
Refresh repo list	`apt update`	`dnf check-update`
Upgrade packages	`apt upgrade`	`dnf upgrade`
Install package	`apt install`	`dnf install`

Task	Debian (APT/Dpkg)	Red Hat (DNF/RPM)
Remove package	`apt remove`	`dnf remove`
Purge configs	`apt purge`	*no direct purge, configs remain unless manual*
Check dependencies	`apt depends`	`dnf deplist`
Reverse deps	`apt rdepends`	`dnf repoquery --whatrequires`
Detailed info	`apt show`	`rpm -qi`

🧪 FIELD TRAINING

1. On **Debian/Ubuntu**, run:

   `apt depends curl`

   → See what libraries `curl` needs.
2. On **Red Hat/CentOS/Fedora**, run:

   `dnf deplist curl`

   → Compare dependencies with Debian.
3. Try removing a package:
   - Debian: `sudo apt purge nano`
   - Red Hat: `sudo dnf remove nano`
     → Observe config files behavior.
4. Update all packages:
   - Debian: `sudo apt full-upgrade`
   - Red Hat: `sudo dnf upgrade`

📎 Final Note

- Repositories = warehouses.
- Packages = units of software.
- Dependencies = invisible chains you must respect.
- **Debian tools (APT/Dpkg)** → powerful with purge/reverse deps.
- **Red Hat tools (DNF/RPM)** → enterprise-grade, slower but very detailed.

🧠 INTELLIGENCE FIELD REPORT: Complete — Linux Storage, Devices & Filesystem Operations (Part 3 — Expanded)

⚠️ MISSION-CRITICAL WARNING (read first)

Many commands below can **destroy data irreversibly** if pointed at the wrong device. Always verify target devices and backups before running destructive commands.

Safe verification workflow BEFORE destructive action:

`# identify devices lsblk -f blkid # confirm content & size sudo fdisk -l /dev/sdX stat -c "%s" file.iso # optionally mount read-only to inspect sudo mount -o ro /dev/sdX1 /mnt/test`

---

1) DEVICE NAMES & MAPPING (quick reference)

Pattern	Device type	Notes
`/dev/sd*`	modern disks (SCSI/SATA/NVMe/USB)	`/dev/sda, /dev/sdb`, partitions `/dev/sdb1`
`/dev/nvme*`	NVMe devices	`/dev/nvme0n1p1` (different naming)
`/dev/hd*`	old PATA/IDE devices	legacy, rare on modern systems
`/dev/sr*`	optical (CD/DVD)	`/dev/sr0`
`/dev/loop*`	loopback devices (mount ISO loop/files)	use `losetup` to manage
`/dev/fd*`, `/dev/lp*`	floppy/printer	legacy

---

2) INSPECTION & IDENTIFICATION (essential commands)

- `lsblk -o NAME,FSTYPE,UUID,LABEL,MOUNTPOINT,SIZE` — visual device tree.
- `lsblk -f` — quick FS/UUID view.
- `blkid /dev/sdX` — print UUID/TYPE/LABEL.

- `udevadm info --query=all --name=/dev/sdX` — device metadata from udev.
- `fdisk -l /dev/sdX` — partition table details (MBR).
- `parted /dev/sdX print` — GPT-aware partition table info.
- `sgdisk -p /dev/sdX` — GPT partitions (gdisk/sgdisk).
- `smartctl -a /dev/sdX` — SMART health (install smartmontools).
- `hdparm -I /dev/sdX` — device identification & capabilities.

---

3) PARTITIONING (tools & safe usage)

- `fdisk /dev/sdX` — MBR/legacy partitioning interactive. Use for BIOS/MBR.
  - `t` changes partition type (only ID).
  - `p` print, `w` write, `q` quit no-write.
- `parted /dev/sdX` — GPT-aware, scriptable (`mklabel` , `mkpart` , `set` flags).
- `gdisk /dev/sdX` — GPT fdisk (like fdisk for GPT).
- `sfdisk` — scriptable partitioning (useful for automation).
- **Wipe signatures (if needed before new partitioning):** `sudo wipefs -a /dev/sdX sudo dd if=/dev/zero of=/dev/sdX bs=1M count=10` *Warning: destructive.*

---

4) CREATING FILESYSTEMS (`mkfs` family)

- `mkfs.ext4 /dev/sdX1` (or `mkfs -t ext4`) — format ext4. Options:
  - `-L <label>` set filesystem label
  - `-m <percent>` reserve space for root (`-m 1` for small disks)
  - `-O` enable/disable features
- `mkfs.xfs /dev/sdX1` — XFS (use `xfs_progs` tools for repair/resize).
- `mkfs.vfat /dev/sdX1` — FAT (for USB compatibility).
- `mkfs.fat -F 32` — force FAT32.
- `mke2fs` / `mkfs.ext3` / `mkfs.ext2` — older ext variants.
- `mkfs.ntfs` or `mkfs.ntfs -f` (ntfs-3g) — NTFS formatting if needed.

---

5) MOUNTING (options & examples)

- Basic mount: `sudo mount /dev/sdb1 /mnt/test`
- Mount read-only: `sudo mount -o ro /dev/sdb1 /mnt/test`

- Common mount options (security & behaviour):
  - `ro`, `rw` — readonly / read-write.
  - `noexec` — prevent running binaries from FS.
  - `nosuid` — ignore setuid bits.
  - `nodev` — ignore device files.
  - `uid=1000,gid=1000` — for vfat/exfat to set owner.
  - `fmask=0111,dmask=0000` — file/dir permission masks for vfat.
  - `noatime`, `relatime` — reduce atime updates (performance).
  - `errors=remount-ro` — extX safety on errors.
- Mount with filesystem type: `sudo mount -t vfat /dev/sdb1 /mnt/usb`
- Mount ISO loop: `sudo mount -o loop,ro ubuntu.iso /mnt/iso # or with losetup sudo losetup /dev/loop0 ubuntu.iso sudo mount /dev/loop0 /mnt/iso`
- `findmnt -o TARGET,SOURCE,FSTYPE,OPTIONS` — inspect mounts.
- `mount -a` — apply `/etc/fstab` entries (test after editing fstab).

---

6) UNMOUNTING & STUCK DEVICES

- Normal unmount: `sudo umount /mnt/test`
- If busy:
  - Inspect processes: `sudo lsof +D /mnt/test sudo fuser -vm /mnt/test`
  - Send kill (use with care): `sudo fuser -km /mnt/test`
- Lazy unmount (unbind but keep until free): `sudo umount -l /mnt/test`
- Force unmount (dangerous): `sudo umount -f /mnt/test` *Avoid unless necessary (NFS/loop issues).*

---

7) FILESYSTEM CHECK & REPAIR

- **ext2/3/4**: `sudo umount /dev/sdX1 sudo fsck -f /dev/sdX1 # auto-yes (dangerous) sudo fsck -y /dev/sdX1`
  - DO NOT run `fsck` on mounted RW filesystem.
- **XFS**: `sudo umount /dev/sdX1 sudo xfs_repair /dev/sdX1 # If log corrupt sometimes: sudo xfs_repair -L /dev/sdX1`
  - `xfs_repair` requires unmounted filesystem (or mount readonly in emergency).
- **FAT/NTFS**:
  - `dosfsck -a /dev/sdX1` — auto repair FAT.

- `ntfsfix /dev/sdX1` — limited NTFS fix; prefer Windows `chkdsk`.

- **Bad block scan** (very slow & destructive risk):
  - `badblocks -v /dev/sdX` — read-only scan
  - Use `-n` for non-destructive test (still risky).
- **resizing**:
  - ext: `resize2fs /dev/sdX1` (after partition change).
  - xfs: cannot shrink easily; to grow use `xfs_growfs` while mounted RW.

---

8) COPY & IMAGING (safe examples)

- `dd` (low-level): `sudo dd if=linux.iso of=/dev/sdb bs=4M status=progress conv=fsync`
  - `conv=fsync` helps flush data; `status=progress` shows progress.
  - Prefer specifying `of=/dev/sdb` not `of=/dev/sdb1` when writing full disk.
- `ddrescue` (recommended for recovery): `sudo apt install gddrescue sudo ddrescue -f -n /dev/sdX image.img mapfile`
  - `ddrescue` is smarter for failing drives (non-destructive attempts).
- Copy files with metadata: `sudo cp -a /source /dest rsync -aAXv /source/ /dest/`

---

9) ISO / CD/DVD SPECIFICS

- ISO sector size: **2048 bytes** (standard CD/DVD).
- Create ISO: `genisoimage -o cd-rom.iso -R -J ~/dir # or mkisofs -o cd-rom.iso -R -J ~/dir`
- Burn ISO to disc (wodim/cdrdao): `wodim dev=/dev/sr0 image.iso # blank wodim dev=/dev/cdrw blank=fast`
- Verify ISO vs disc: `md5sum dvd-image.iso dd if=/dev/sr0 bs=2048 count=$(( $(stat -c "%s" dvd-image.iso) / 2048 )) | md5sum`

---

10) SWAP MANAGEMENT

- Show swap: `swapon --show`
- Enable/disable swap: `sudo swapon /dev/sdX3 sudo swapoff /dev/sdX3`
- `swapon -a` reads fstab and enables swaps.
- Swap can be file or partition.

11) MOUNT OPTIONS FOR NON-UNIX FS (practical)

- `vfat` / `exfat`: `sudo mount -t vfat -o uid=1000,gid=1000,fmask=113,dmask=002 /dev/sdb1 /mnt/usb`
- `iso9660` (CD images) set `uid` if needed.
- `ntfs-3g` for NTFS read/write: `sudo mount -t ntfs-3g /dev/sdb1 /mnt/ntfs -o uid=1000,gid=1000`

---

12) UUIDs, LABELs & fstab best practices

- Get UUID: `sudo blkid -s UUID -o value /dev/sdb1`
- fstab entry (recommended use UUID): `UUID=xxxx-xxxx /data ext4 defaults,noatime,errors=remount-ro 0 2`
- After editing `/etc/fstab`, always test: `sudo mount -a`
  - Keep a root shell open or remote session fallback when testing fstab on remote machines.

---

13) LOOP DEVICES & IMAGE MANAGEMENT

- Setup loop and mount: `sudo losetup --find --show ubuntu.iso sudo mount /dev/loop0 /mnt/iso`
- Detach: `sudo umount /mnt/iso sudo losetup -d /dev/loop0`

---

14) ADVANCED: namespaces, systemd, automounting

- `systemd-mount` / `systemd-automount` — ephemeral/system-managed mounts (useful on modern distros).
- Mount namespaces (`unshare --mount`) for sandboxed mounts — advanced use.

---

15) RECOVERY & DIAGNOSTICS TOOLS

- `testdisk` — recover partitions and filesystems.
- `photorec` — file carving (recovers files by type).

- `gpart` — undelete/recover partitions (older).
- `smartctl` — S.M.A.R.T. diagnostics for drive health.
- `badblocks` — scan for bad sectors.
- `tune2fs -l /dev/sdX1` — show ext filesystem parameters.
- `debugfs` — ext filesystem low-level debug and recovery (dangerous—read manual).

---

16) PRACTICAL EXAMPLES (copy-paste ready)

1. **Identify & mount USB read-only**

   `lsblk -f sudo mount -o ro /dev/sdb1 /mnt/usb`
2. **Safely write ISO to USB (USB as installer)**

   `sudo dd if=ubuntu.iso of=/dev/sdb bs=4M status=progress conv=fsync sync`
3. **Create & mount an ext4 on partition**

   `sudo mkfs.ext4 -L MyData /dev/sdb1 sudo mount -o noexec,nosuid,nodev /dev/sdb1 /mnt/data`
4. **Check & repair ext4**

   `sudo umount /dev/sdb1 sudo fsck -f /dev/sdb1`
5. **Verify ISO vs DVD**

   `md5sum dvd-image.iso dd if=/dev/sr0 bs=2048 count=$(( $(stat -c "%s" dvd-image.iso) / 2048 )) | md5sum`

---

17) TROUBLESHOOTING COMMON SCENARIOS

- **Device not found after plug**: `dmesg | tail` → check kernel messages.
- **"device is busy" on umount**: use `lsof +D` / `fuser -vm` to find processes.
- **Mount shows wrong owner on vfat**: use `uid=,gid=` options.
- **I/O errors / SMART failing**: `smartctl -a /dev/sdX` → prepare to replace drive.

---

18) SECURITY & HARDENING RECOMMENDATIONS

- Mount removable media with `noexec,nosuid,nodev`.
- Use `errors=remount-ro` for important ext partitions to avoid silent corruption.

- Avoid `auto` mount for untrusted devices; require manual mounts.
- Regular SMART checks and backups are mandatory for critical systems.

---

19) FIELD EXERCISES (measurable)

1. Plug a USB, run `lsblk -f` and `blkid`, mount it read-only, list files, then unmount. (Record commands & output.)
2. Create `test.img` (1G), format ext4, mount loop, write a file, unmount, check image size & `stat`.

   `dd if=/dev/zero of=test.img bs=1M count=1024 mkfs.ext4 test.img sudo losetup --find --show test.img sudo mount /dev/loop0 /mnt/tmp echo "ok" | sudo tee /mnt/tmp/test.txt sudo umount /mnt/tmp sudo losetup -d /dev/loop0`
3. Make an ISO of a small folder, calculate `md5sum`, then simulate DVD read `dd` with `bs=2048` & compare checksums.

Final Tactical Summary

- Device naming is critical — `lsblk` and `blkid` are your first lines of defense.
- Partition with `fdisk/parted/gdisk` depending on MBR/GPT.
- Format only after confirming targets — `mkfs` wipes data.
- Use `fsck/xfs_repair/dosfsck/ntfsfix` appropriately; **never** on mounted RW FS.
- `dd` is powerful but dangerous — prefer `ddrescue` for failing media.
- For portability of prompts and scripts use UUIDs in `fstab`.
- Always mount removable media with secure options (`noexec,nosuid,nodev`).
- Maintain backups and run SMART checks regularly. (stat /location on mount point --> good than lsblk if you want block one)

page 213-241

1) Networking toolkit (quick hits)

- **Ping** — connectivity probe.
- **tracepath / traceroute** — map route, detect hop latency.
- **ip** — interface/address/status control (`ip a` , `ip route`).
- **netstat** (or `ss`) — sockets, listening ports, connections.
- **sftp, scp** — file transfer over SSH.
- **wget** — non-interactive HTTP/FTP fetch.

Notes:

- We trained on **SSH / OpenSSH**: client (`ssh` , `sftp` , `scp`) and server (`sshd`).
- **Windows → Linux via SSH**: use **PuTTY** (or Windows built-in `ssh` on modern Windows).
- Key management: generate keys (`ssh-keygen -t ed25519 -f ~/.ssh/id_ed25519`) and copy public key (`ssh-copy-id` or append `~/.ssh/id_ed25519.pub` to `~/.ssh/authorized_keys`).

---

2) Fast file discovery — `locate` vs `find`

- **`locate`** — instant lookup via DB: `/var/lib/mlocate/mlocate.db` .
  - Update DB: `sudo updatedb` (cron/systemd usually does it).
  - Pros: fast. Cons: may be stale (reflects DB snapshot).
- **`find`** — authoritative, live filesystem traversal.
  - Syntax: `find <path> <tests> <actions>`
  - Tests decide *which* files; Actions decide *what* to do.

---

3) `find` — Tests (common)

- `-type f` — regular files.
- `-type d` — directories.
- `-name "pattern"` — case-sensitive name match.
- `-iname "pattern"` — case-insensitive.
- `-size` , `-perm` , `-user` , `-group` , `-empty`

- `-mtime/-ctime/-mmin/-cmin` — time filters.
- `-nouser` / `-nogroup` — orphaned objects (useful for incident triage).
- `-newer <file>` — files modified after `<file>` (useful as a timestamp marker).

---

4) `find` — Actions (common)

- `-print` — print path (default if no action specified).
- `-ls` — long listing.
- `-delete` — delete (implies `-depth`). **Dangerous** — use with care.
- `-exec cmd '{}' \;` — run `cmd` per file (one system call per file).
- `-exec cmd '{}' +` — batch files into one `cmd` invocation (much faster).
- `-ok` / `-ok ... +` — interactive `-exec` (asks permission).

---

5) Logical operators & precedence

- `-and` (or implicit space) — both tests must pass.
- `-or` (`-o`) — either test suffices.
- `-not` (`!`) — negate test.
- Use `\(...\)` to group expressions and control evaluation.

Short-circuit rules:

- `A -and B`: if A is **False**, B is **not evaluated**.
- `A -or B`: if A is **True**, B is **not evaluated**.
  (Leverage this for performance on large trees.)

---

6) Practical `find` patterns (tactical examples)

- Files not 0600 or dirs not 0700 in `$HOME`: `find "$HOME" \( -type f ! -perm 0600 -o -type d ! -perm 0700 \) -print`
- Files changed after marker: `touch /tmp/marker && find /data -type f -newer /tmp/marker`
- Fix permissions in place (non-printing, destructive): `find playground \( -type f ! -perm 0600 -exec chmod 0600 '{}' \; \) -o \( -type d ! -perm 0700 -exec chmod 0700 '{}' \; \)`

7) Traversal options (control search bounds)

- `-depth` — process directory contents before the directory itself (required by `-delete`).
- `-maxdepth N` — do not descend below N levels.
- `-mindepth N` — start applying tests/actions only at depth ≥ N.
- `-mount` (a.k.a. `-xdev`) — do not traverse into other mounted filesystems.
- `-noleaf` — disable Unix leaf optimization (use when scanning non-Unix filesystems like FAT or ISO9660).

---

8) `xargs` — why and how

- `xargs` builds command lines from stdin; reduces fork/exec overhead compared to `-exec \;`.
- Use **null-separated** mode for safety with weird filenames: `find . -iname '*.jpg' -print0 | xargs -0 ls -l find . -iname '*.jpg' -print0 | xargs -0 rm -f`
- Mismatch risks: `-print0` must pair with `xargs -0` (or `--null`). Missing pairing → broken behavior or no action.

---

9) File & permission ops — quick rules

- `.ssh` hardening: `chmod 700 ~/.ssh chmod 600 ~/.ssh/authorized_keys ~/.ssh/id_*`
- Review special bits: `find / -perm /6000 -type f -ls # find suid/sgid files`
- Key generation: `ssh-keygen -t ed25519 -C "you@host" -f ~/.ssh/id_ed25519`

---

10) SOPs & safety checklist (must follow)

1. **Test first** — run `find ... -print` before `-exec` or `-delete`.
2. **Use `-print0 | xargs -0`** with untrusted filenames.
3. **Prefer `-exec ... +` or `xargs`** for bulk actions (performance).
4. **Use `-mount`** if you must avoid crossing filesystems.
5. **Backup or snapshot** before mass `chmod` / `chown` / `rm`.
6. Confirm `sshd` changes with `sshd -t` before `systemctl reload sshd`.

One-page checklist (field cheat)

- Keygen: `ssh-keygen -t ed25519 -f ~/.ssh/id_ed25519`
- Copy key: `ssh-copy-id user@host`
- Locate fast: `locate filename` → `sudo updatedb` if stale
- Live search: `find /path -type f -name 'pat*' -print0 | xargs -0 ls -l`
- Find recent: `find /path -newer /tmp/marker`
- Fix perms `.ssh`: `chmod 700 ~/.ssh && chmod 600 ~/.ssh/*`
- Check suid/sgid: `find / -perm /6000 -type f -ls`
- Test sshd config: `sudo sshd -t && sudo systemctl reload sshd`

page 270-290

🛰️ [FIELD DOSSIER] — Text Processing & Regular Expressions Ops

> **Clearance:** OPERATIONAL
> **Operative:** Cybryon
> **Purpose:** Tactical reference for advanced text filtering, sorting, deduplication, and field extraction using GNU text utilities (`grep` , `sort` , `uniq` , `cut`) with operational awareness of regex logic.

🎯 ONE-LINE SUMMARY

`grep` locates patterns via regex; `sort` organizes lines or datasets; `uniq` removes or counts duplicates; `cut` extracts precise data fields — together they form the core of Unix-grade text intelligence and forensic log analysis.

🧠 SECTION A — `grep` OPERATIONS (Pattern Recognition Unit)

Mission Purpose

`grep` scans input streams or files for patterns defined by **regular expressions (regex)**. Used for log triage, data extraction, and validation.

Core Syntax

```
grep [OPTIONS] PATTERN [FILE...]
```

Essential Options

Option	Long Form	Description
`-i`	`--ignore-case`	Case-insensitive matching
`-v`	`--invert-match`	Show non-matching lines
`-E`	`--extended-regexp`	Enable Extended Regex (ERE)

Option	Long Form	Description
`-P`	`--perl-regexp`	Use PCRE (advanced syntax)
`-c`	`--count`	Show number of matching lines
`-n`	`--line-number`	Prefix line numbers
`-l`	`--files-with-matches`	List filenames that match
`-L`	`--files-without-match`	List filenames without matches
`-h`	`--no-filename`	Suppress filenames in multi-file output

Tactical Regex Reference

Symbol	Meaning	Example
`.`	Any single char	`grep 'h.llo' file`
`^`	Line start	`grep '^root' /etc/passwd`
`$`	Line end	`grep 'bash$' file`
`[]`	Character class	`[A-Z]`, `[[:digit:]]`
`[^]`	Negated class	`[^0-9]` non-digits
`+`	One or more	`([[:alpha:]] ?)+`
`{n}`	Exact repetition	`[0-9]{3}`
`` ` ``	`` ` ``	Alternation

Operational Examples

```
# Detect uppercase letters echo "heLlo world" | grep -E '[[:upper:]]' # Upper followed by any lower/upper echo "heLlo world" | grep -E '[[:upper:]][[:lower:][:upper:]]' # Validate phone format: (###) ####-#### echo "(255) 4346-5226" | grep -E '^\([0-9]{3}\) [0-9]{4}-[0-9]{4}$' # Detect malformed numbers (exclude correct ones) cat phon.txt | grep -Ev '^\([0-9]{2}\)$' # Match lines ending with .txt grep -E '\.txt$' list.txt
```

Note: `grep` uses **ERE** when `-E` is enabled, while `vim` search and `sed` defaults use **BRE**, requiring extra backslashes (e.g. `\{3\}`).

⚙️ SECTION B — `sort` OPERATIONS (Order & Classification)

Purpose

Arranges text lines or dataset entries in lexical or numeric order — vital for normalization before further parsing or deduplication.

Basic Syntax

```
sort [OPTIONS] [FILE...]
```

Useful Options

Option	Long	Description
`-b`	`--ignore-leading-blanks`	Ignore leading spaces
`-f`	`--ignore-case`	Case-insensitive sort
`-n`	`--numeric-sort`	Sort numerically
`-r`	`--reverse`	Reverse order
`-k`	`--key=field1[,field2]`	Sort by key field(s)
`-m`	`--merge`	Merge presorted files
`-o`	`--output=file`	Output to file
`-t`	`--field-separator=char`	Define field separator

Operational Example

Given file `linux.txt`:

```
Fedora 10 11/25/2008 Fedora 5 03/20/2006 Ubuntu 8.10 10/30/2008 SUSE 10.3 10/04/2007
```

Sort by second field (numeric version):

```
sort -k 2n linux.txt
```

Sort alphabetically by distribution:

```
sort -k 1 linux.txt
```

Merge two sorted lists:

```
sort old.txt new.txt > merged.txt
```

🧩 SECTION C — `uniq` OPERATIONS (Duplication Filter)

Mission Purpose

Removes or identifies duplicate adjacent lines — **requires sorted input** for proper operation.

Core Syntax

```
uniq [OPTIONS] [FILE]
```

Key Options

Option	Long	Description
`-c`	`--count`	Prefix duplicates with occurrence count
`-d`	`--repeated`	Show only repeated lines
`-u`	`--unique`	Show unique lines only
`-i`	`--ignore-case`	Case-insensitive comparison
`-f n`	`--skip-fields=n`	Ignore first n fields
`-s n`	`--skip-chars=n`	Ignore first n characters

Operational Example

```
# Sort, then remove duplicates sort game.txt | uniq # Show count of each repeated line sort game.txt | uniq -c
```

Output example:

```
3 a 1 b 2 c
```

→ `a` occurred three times.

🧱 SECTION D — `cut` OPERATIONS (Field Extraction Unit)

Mission Purpose

Extract specific columns or character ranges from structured text (logs, CSVs, or output streams).

Syntax

`cut [OPTIONS] FILE`

Key Options

Option	Long	Description
`-c list`	`--characters=list`	Extract character positions or ranges
`-f list`	`--fields=list`	Extract fields
`-d char`	`--delimiter=char`	Specify field delimiter (default: TAB)
`--complement`		Extract everything *except* the specified parts

Tactical Examples

`# Extract 3rd field separated by spaces cut -d ' ' -f 3 linux.txt # Extract 4th character from that result cut -d ' ' -f 3 linux.txt | cut -c 4`

If the result is `2009/12/1`, output will be:

`9`

Tip: Use `expand` to convert tabs to spaces for precise alignment during analysis.

🧭 FIELD TESTS / PRACTICE OPS

1. **Grep Intelligence Sweep**

   `echo -e "Alpha\nbeta\nGamma" > sample.txt grep -E '^[[:upper:]]' sample.txt`
2. **Sorting Dataset**

   `sort -k 2n linux.txt > ordered.txt`
3. **Duplicate Elimination**

   `sort data.txt | uniq -c`
4. **Field Extraction**

   `cut -d ':' -f 1 /etc/passwd | sort | uniq | less`

⚠️ OPERATIONAL WARNINGS

- Always **sort before using** `uniq` for accurate duplicate detection.
- Use **quotes around regex** to prevent shell expansion.
- `cut` misreads multi-space columns unless proper delimiters are defined.
- Escaping rules differ between **BRE** (`grep`) and **ERE** (`grep -E` / `egrep`).
- `sort -n` may mis-order mixed text/numeric lines — normalize data first.

🧭 QUICK REFERENCE CHEAT SHEET

Tool	Primary Function	Example
`grep -E`	Match patterns (ERE)	`grep -E '[[:upper:]]' file`
`sort`	Order lines	`sort -k 2n file`
`uniq`	Remove duplicates	`sort file
`cut`	Extract fields	`cut -d ' ' -f 3 file`

part 1 !

page 290-310

🧩 Text Processing & Comparison Commands (Deep Summary)

🧠 sort

```
sort -k 3.7nbr -k 3.1nbr -k 3.4nbr distros.txt > distros-by-date.txt
```

Explanation:
This command sorts the file `distros.txt` based on multiple sorting keys.

- **`-k 3.7nbr`** →
  - `-k` : sort by a specific *field* (split by spaces by default).
  - `3.7` : start at character 7 in field 3.
  - `n` : numerical sort.
  - `b` : ignore leading spaces.
  - `r` : reverse the result (descending order).

If values in `-k 3.7nbr` are equal, sorting continues with `-k 3.1nbr`, then `-k 3.4nbr`.

✂️ cut

```
cut -f 1,2 distros-by-date.txt > distros-versions.txt
```

Explanation:
Print only fields 1 and 2 into a new file.

> ⚠️ On your system, you must specify the field delimiter:
> `cut -d ' ' -f 1,2 distros-by-date.txt`

🔗 paste

```
paste file1.txt file2.txt
```

Explanation:
Combines files *horizontally*, merging their corresponding lines field by field.

🔗 join

```
join -1 2 -2 2 file1.txt file2.txt
```

Explanation:
Merges lines based on a *common field*.

- `-1 2` → use field 2 from file1
- `-2 2` → use field 2 from file2

⚖️ comm

```
comm file1.txt file2.txt
```

Explanation:
Compares two **sorted** files line by line, producing three columns:

1. Unique lines in file1
2. Unique lines in file2
3. Lines common to both

⚔️ diff

```
diff -u file1.txt file2.txt
```

Explanation:
Shows the *differences* between two files.

- `-` → lines removed
- `+` → lines added
- `(space)` → unchanged lines

> `-u` (unified format) is preferred, especially for Git or collaborative patches.

🩹 patch

```
diff -Naur file1.txt file2.txt > patchfile.txt
```

Options:

- `N` → Treat missing files as empty
- `a` → Treat all files as text
- `u` → Unified diff format
- `r` → Compare directories recursively

Apply the patch with:

```
patch < patchfile.txt
```

This modifies `file1.txt` based on the differences stored in `patchfile.txt`.

🔠 tr

Transliteration and deletion utility

Examples:

```
echo "hello world" | tr [a-z] [A-Z] # Output: HELLO WORLD
```

```
tr -d '\r' < winfile.txt > linuxfile.txt
```

Removes Windows `\r` (carriage return), converting `\r\n` → `\n`.

ROT13 Example (simple cipher):

```
echo "hello" | tr a-z n-za-m # Output: uryyb
```

🧩 sed (Stream Editor)

🔄 Basic substitution

```
echo "hello world" | sed 's/hello/bob/' # Output: bob world
```

Use `_` instead of `/` if needed:

```
sed 's_hello_bob_'
```

📜 Conditional printing and deletion

```
sed -n '/hello/p' file.txt # print only lines containing 'hello' sed -n
'/hello/!p' file.txt # print all lines except those with 'hello' sed '/error/d'
```

`file.txt # delete lines containing 'error'`

Without `-n`, `sed` prints every line *plus* the matched results again — so `-n` is often used for controlled output.

📖 Addressing patterns

Pattern	Meaning	Example
`5`	Specific line	`5d` → delete line 5
`$`	Last line	`$p` → print the last line
`/regex/`	Lines matching pattern	`/error/d`
`1,5`	From line 1 to 5	`1,5p`
`/start/,/end/`	From pattern to pattern	`/BEGIN/,/END/p`
`1~2`	Every second line starting at 1	`1~2d` deletes odd lines
`/match/,+3`	Line matching + next 3	`/root/,+3p`
`/match/!`	Everything except match	`/error/!p`

⚙️ sed Commands Reference

Command	Meaning	Example	Result
`=`	Print line number	`sed '=' file.txt`	Shows line numbers
`a`	Append after match	`/err/a\New`	Adds text after
`i`	Insert before match	`/err/i\New`	Adds text before
`d`	Delete line	`/tmp/d`	Deletes
`p`	Print line	`-n '/err/p'`	Conditional print
`q`	Quit and print	`3q`	Stop at line 3
`Q`	Quit silently	`3Q`	Stop, no print
`s///`	Substitution	`s/foo/bar/g`	Replace all
`y///`	Transliteration	`y/abc/xyz/`	Simple cipher

🧠 Deep Example: Date format conversion

Convert **MM/DD/YYYY → YYYY-MM-DD**:

```
sed 's/\([0-9]\{2\}\)\/\([0-9]\{2\}\)\/\([0-9]\{4\}\)$/\3-\1-\2/' distros.txt
```

🧰 sed Script Example

File: `script.sed`

```
# sed script to produce Linux distributions report 1 i\ \ Linux Distributions Report\ s/\([0-9]\{2\}\)\/\([0-9]\{2\}\)\/\([0-9]\{4\}\)$/\3-\1-\2/ y/abcdefghijklmnopqrstuvwxyz/ABCDEFGHIJKLMNOPQRSTUVWXYZ/
```

Explanation:

- Adds two blank lines and inserts the title *before the first line*.
- Converts date format.
- Translates lowercase → uppercase.

Run with:

```
sed -f script.sed distros.txt
```

✍️ aspell

Used for **spell checking**:

```
aspell check wrong.txt
```

Example content:

```
hii cats we mett again!
```

`aspell` will interactively suggest corrections for each misspelled word (`hii` , `mett`), allowing you to pick the right one by number.

page 310-350

Here's the **deep, professional summary** for your second part (covering `nl`, `fold`, `fmt`, `pr`, `printf`, `groff`, `make`, and `gcc`), written in the **exact same structured style** as your reference, but with deeper technical analysis and conceptual linking to real software compilation workflows.

⚙️ Text Formatting, Reporting & Source Compilation (Deep Summary)

🔢 nl — Number Lines Intelligently

Syntax:

```
nl [options] file
```

Purpose:
Adds line numbers to text files — but unlike `cat -n`, it **distinguishes between header, body, and footer** sections.

Key Concept:
`nl` treats the file as having three possible regions:

- **Header:** marked by `:::`
- **Body:** marked by `::`
- **Footer:** marked by `:`

To make `nl` recognize them literally, you must **escape colons**:

```
:\:\:\ # Escaped ::: means “start of header section”
```

Comparison with `cat -n`:

Command	Behavior	Custom Sections	Blank Line Handling
`cat -n`	Numbers every line	❌ No distinction	Numbers empty lines
`nl`	Numbers only the “body”	✅ Supports header/body/footer	Skips empty lines by default

🪚 fold — Hard Line Wrapping

Syntax:

```
fold -w <width> [-s] file
```

Purpose:
Wraps each line to a specified **character width**, useful when formatting text output for terminals or reports.

Examples:

```
fold -w 10 file.txt
```

→ Splits lines every 10 characters, even if words break.

With `-s` ("smart split"):

```
fold -w 10 -s file.txt
```

→ Splits only at word boundaries.

Use Case:
When sending log outputs or reports to fixed-width terminals, `fold` ensures no line exceeds a certain width.

🧠 fmt — Intelligent Text Formatter

Syntax:

```
fmt [-w width] [-p prefix] file
```

Purpose:
Formats text **paragraphs intelligently** — not just by width, but preserving indentation, word boundaries, and optional comment prefixes.

Example:

```
fmt -w 10 -p '#'
```

→ Rewraps lines to width 10, but keeps each line starting with `#`.

Use Case:
Ideal for source code comments, documentation, and commit messages — keeps code

comments aligned and readable.

Difference vs. `fold`:

Tool	Logic	Word Awareness	Prefix Support
`fold`	Purely by width	❌	❌
`fmt`	Semantic reflow	✅	✅ (`-p`)

🗒️ pr — File Pagination and Reporting

Syntax:

```
pr [options] file
```

Purpose:
Prepares a file for printing — adds headers, page numbers, and column formatting.

Example:

```
pr -l 15 -w 50 file.txt
```

→ Displays file as pages of **15 lines each**, **50 characters wide**.

Use Case:
When producing formatted terminal reports — e.g., system audit summaries, research logs, or print-ready text layouts.

Notable Options:

Option	Meaning
`-l <n>`	Lines per page
`-w <n>`	Page width
`-h <text>`	Custom header text
`-d`	Double spacing

🧩 printf — Structured and Formatted Output

Syntax:

```
printf "format" [arguments...]
```

Purpose:
Outputs formatted data — much more reliable than `echo`.

Example:

```
printf "hello %s\n" "world"
```

→ Prints `hello world` followed by a newline.

Common Format Specifiers:

Symbol	Meaning	Mnemonic
`%d`	Decimal integer	(D = Decimal)
`%f`	Floating point	(F = Float)
`%o`	Octal	(O = Octal)
`%s`	String	(S = String)
`%x`	Hex (lowercase)	(x = hex)
`%X`	Hex (uppercase)	(X = HEX)
`%%`	Literal `%`	Escape

Deep Example:

```
printf "Int: %d | Float: %.2f | Octal: %o | Hex: %X | String: %s | Percent: %%\n" \ 42 3.1415 42 42 "Linux"
```

Output:

```
Int: 42 | Float: 3.14 | Octal: 52 | Hex: 2A | String: Linux | Percent: %
```

Usage Context:
`printf` is critical in scripting for **consistent output** (e.g., logging, color formatting, data alignment).

🖋️ groff — GNU Document Formatter

Purpose:
Formats text documents using the **roff** typesetting language (used for `man` pages).

Key Insight:
`groff` is behind almost every `man` page you read — it converts `.man` or `.roff` source files into terminal-friendly or print-ready text.

Example:

```
zcat /usr/share/man/man1/ls.gz | groff -mandoc -T ascii | head
```

- `zcat` : decompresses `.gz` man page
- `groff` : formats it
- `-mandoc` : use man macros
- `-T ascii` : output as terminal text (instead of PostScript or PDF)

Output:
Displays a perfectly formatted man page identical to `man ls` .

Use Case:
For customizing documentation, converting `.man` files into readable or printable text.

🧱 gcc — GNU Compiler Collection

Purpose:
Translates **C source code (`.c`) into machine code (`.o` , `.out`)**.

⚙️ Compilation Pipeline

1. **Preprocessing:** expands macros, removes comments
   → Output: pure C code
   Command: `gcc -E file.c`
2. **Compilation:** converts C to assembly
   → `gcc -S file.c`
3. **Assembly:** converts assembly to object code
   → `gcc -c file.c` → produces `file.o`
4. **Linking:** merges all `.o` files into an executable
   → `gcc file1.o file2.o -o program`

Example:

```
gcc -c hello.c # Produces hello.o gcc -c donate.c # Produces donate.o gcc hello.o donate.o -o script ./script # Run the final binary
```

Conceptual Summary:

Stage	Input	Output	Description
Preprocessor	`.c`	`.i`	Expands macros/includes
Compiler	`.i`	`.s`	Translates to assembly
Assembler	`.s`	`.o`	Converts to machine code
Linker	`.o`	binary	Merges & resolves symbols

🧰 make — Automated Build Manager

Purpose:
Automates compilation by running only the **necessary** commands when files change.

Core Principle:
`make` reads a **Makefile**, which defines:

- Source files
- Dependencies
- Commands to build targets

Example Makefile:

```
script: hello.o donate.o gcc hello.o donate.o -o script hello.o: hello.c gcc -c hello.c donate.o: donate.c gcc -c donate.c clean: rm -f *.o script
```

Usage:

```
make # Builds only what changed make clean # Removes temporary build files
```

Advantages:
✅ Avoids redundant recompilation
✅ Scales to hundreds of files
✅ Foundation of all modern build systems (`cmake` , `ninja` , etc.)

🧩 Integration Insight

The journey from text formatting to code compilation represents the **evolution of command-line mastery**:

Layer	Tool	Purpose
Text formatting	`nl`, `fold`, `fmt`, `pr`, `printf`	Prepare structured readable data
Documentation	`groff`	Format readable technical manuals
Source compilation	`gcc`, `make`	Transform readable code → executable binaries

Each layer refines text — whether it’s a sentence, a paragraph, or a source program — into a **structured and executable form**.

page 350-380

🧩 Part IV — Writing Shell Scripts (Core Principles)

⚙️ 1. Creating Executable Shell Scripts

A shell script is simply a **text file containing shell commands**, with execution permissions enabled.

Example:

```
#!/bin/bash echo "Hello world!"
```

Steps to Run:

1. **Create the file:**

   ```
   vim hello.sh
   ```
2. **Make it executable:**

   ```
   chmod +x hello.sh
   ```
3. **Place it in your `$PATH`** (e.g., `/usr/local/bin`) so you can execute it directly:

   ```
   hello.sh
   ```

Concept:
The **shebang (`#!`)** at the top tells the system which interpreter to use — here `/bin/bash`.

💾 2. Variables — Simple and Expandable

Example:

```
#!/bin/bash a=5 b=10 var="Hello world! We are good. It's $(date +'%x') now! $((a + b))" echo "$var"
```

> ⚠️ **Important:** No spaces around `=` during assignment.

Alternative Output Method (Using `cat << END`)

To avoid repetitive `echo` statements:

`cat << END $var END`

This **here-document syntax** allows multi-line text and variable expansion — useful for formatted outputs.

🌐 3. Automating FTP Sessions

Example:

```
#!/bin/bash server='test.rebex.net' serverLogin='demo password' ftp -n << POP
open $server user $serverLogin get file_name bye POP
```

Explanation:

- `ftp -n` : prevents auto-login prompts.
- Credentials are supplied manually via `user $serverLogin` .
- The block between `<< POP` and `POP` is treated as **input stream** (here-document) for the `ftp` command.

Use Case:
Automating repetitive FTP transfers (downloads, uploads) without user interaction.

🎨 4. Syntax Highlighting in Vim

To enable syntax highlighting while editing shell scripts:

```
:syntax on
```

> Useful for distinguishing commands, variables, and strings visually.

🔒 5. Constant (Read-Only) Variables

To define variables that **cannot be modified** later:

```
declare -r var="immutable_value" # or equivalently: readonly
var="immutable_value"
```

Any attempt to reassign such variables will raise an error.

🧱 6. Defining and Using Functions

Syntax:

```
function_name { commands return }
```

Invocation:
Simply call it by name:

```
function_name
```

Example:

```
#!/bin/bash car=10 pop() { local car=5 echo "Inside function: $car" return }
echo "Before: $car" pop echo "After: $car"
```

Output:

```
Before: 10 Inside function: 5 After: 10
```

Concept:

- **`local`** restricts variable scope to the function — it disappears after return.
- Without `local`, changes persist globally, overwriting the external variable.

🧠 7. Key Takeaways

Concept	Description	Command/Example
Shebang (`#!`)	Defines interpreter	`#!/bin/bash`
Execution Permission	Makes script runnable	`chmod +x script.sh`
PATH Location	Run without `./` prefix	Move to `/usr/local/bin`
Variable Expansion	Inline calculations or date	`$(date)`, `$((a+b))`
Here Document	Multi-line formatted text	`cat << END ... END`
FTP Automation	Non-interactive transfers	`ftp -n << POP ... POP`
Syntax Highlighting	Vim command	`:syntax on`
Constant Variable	Immutable declaration	`declare -r var=value`
Local Variables	Scoped within function	`local var=value`

page 380-400

🧠 Organized Summary — Bash If Statements

📌 1️⃣ Basic Structure

`if [ condition ]; then # true else # false fi`

Example:

`if [ x -eq 5 ]; then echo "x is 5" else echo "x is not 5" fi`

📌 2️⃣ File Test Operators → `test [ ]`

`if [ -e filename ]; then echo "File exists" fi`

Common file tests

Expression	Meaning	Example	Notes
`file1 -ef file2`	Same inode (same physical file)	`[ a -ef b ]`	Detects hard links
`file1 -nt file2`	file1 is newer	Depends on timestamps	
`file1 -ot file2`	file1 is older	Opposite of `-nt`	
`-b file`	Block device	`/dev/sda`	Storage devices
`-c file`	Character device	`/dev/tty0`	Terminals
`-d file`	Directory	`/etc`	
`-e file`	Exists (any type)	Most generic	
`-f file`	Regular file	shell scripts, docs	
`-L file`	Symbolic link	Checks link itself	
`-r file`	Read permission		
`-s file`	Size > 0		
`-w file`	Write permission		
`-x file`	Executable permission		

> Best practice: ALWAYS quote variables → `"$file"`

📌 3️⃣ String Expressions

Expression	True if:
`[ string ]`	Not empty
`[ -n string ]`	length > 0
`[ -z string ]`	length == 0
`[ a = b ]` or `[ a == b ]`	Equal
`[ a != b ]`	Not equal
`[ a \< b ]`	a comes before b alphabetically
`[ a \> b ]`	a comes after b alphabetically

⚠️ `<` and `>` must be escaped in `[ ]` to avoid shell redirection

---

📌 4️⃣ Integer Expressions

Expression	Meaning
`-eq`	equal
`-ne`	not equal
`-lt`	less than
`-le`	less or equal
`-gt`	greater than
`-ge`	greater or equal

Example:

```
if [ "$x" -gt 10 ]; then echo "Big number" fi
```

📌 5️⃣ Differences: `[ ]` vs `[[ ]]`

Feature	`[ ]`	`[[ ]]`
Supports regex `=~`	❌	✔️
Safe with spaces in variables	❌	✔️
Supports `&&` / `` ` ``		`/` !` easily
Word splitting risk	High	Low

Example:

```
[[ $name =~ ^[A-Za-z]+$ ]]
```

📌 6️⃣ Arithmetic Expressions — `(( ))` vs `$(( ))`

Syntax	Purpose
`(( expr ))`	Arithmetic condition
`$(( expr ))`	Arithmetic expansion (returns result)

Example:

```
(( count++ )) sum=$(( 10 + 20 ))
```

📌 7️⃣ Command Substitution: `$()` vs Arithmetic: `$(( ))`

Expression	Meaning
`$()`	Execute command inside command
`$(( ))`	Perform math

Examples:

```
files=$(ls) num=$(( 4 * 5 ))
```

📌 8️⃣ `exit` vs `return`

Command	Affects
`exit`	Ends the whole script
`return`	Ends only a function

Exit status:

- `0` → success
- `1` → failure (or any non-zero)

📌 9️⃣ `read` — User Input

Basic:

```
read value
```

With echo:

```
echo "You entered: $value"
```

Useful flags for read

Flag	Description	Example
`-e`	Command-line editing	`read -e var`
`-p "text"`	Prompt message	`read -p "Enter name:" n`
`-t N`	Timeout in seconds	`read -t 5 var`
`-s`	Silent (passwords)	`read -s pass`
`-a`	Store input into array	`read -a arr`
`-n N`	Read only N characters	`read -n 1 key`
`-r`	Raw input, ignore escape `\`	`read -r path`

---

page 400-415

📌 1️⃣ IFS — Internal Field Separator

Used to **split input into fields** when reading strings or files.

Default: space / tab / newline

```
echo "$IFS" # usually prints space or appears empty
```

Temporary override using `read`

Example: parsing a user entry from `/etc/passwd` format

```
line="ahmed:244:545" IFS=":" read user uid gid <<< "$line" echo "$user" # ahmed
echo "$uid" # 244 echo "$gid" # 545
```

✔️ `<<<` provides a **single string** to read
✔️ IFS only affects **this read command**

Changing IFS permanently (with backup)

```
backup_ifs="$IFS" IFS=":" # commands that depend on ':' IFS="$backup_ifs" #
restore original
```

- Best practice: always **restore IFS**, because breaking IFS breaks the entire shell behavior.

IFS with files or pipelines

```
while IFS=":" read f1 f2 f3; do echo "$f1 / $f2 / $f3" done < file.txt
```

or:

```
echo -e "dog:cat:fox\nred:green:blue" | \ while IFS=":" read a b c; do echo
"$b" done
```

📌 2️⃣ Input checking with `[[ ]]`

Examples: validate number / float / string

```
if [[ $var =~ ^[0-9]+$ ]]; then echo "Integer" elif [[ $var =~ ^[0-9]+\.[0-9]+$ ]]; then echo "Float" elif [[ $var =~ ^[A-Za-z]+$ ]]; then echo "String" else echo "Invalid" fi
```

✔️ `[[ ]]` supports regex → best for validation

📌 3️⃣ Loops — `while`

Repeats **while condition is true**

Example:

```
count=1 while [[ "$count" -le 5 ]]; do echo "$count" count=$((count+1)) done
```

---

while with break / continue

```
while true; do echo "Hello!" read -p "Enter a number: " p if [[ "$p" = 0 ]]; then echo "Done!" break else continue fi done
```

✔️ `break` → exit the loop
✔️ `continue` → skip to next iteration

📌 4️⃣ Loops — `until`

Opposite of `while`
→ Runs **until condition becomes true**

```
p=5 until [[ "$p" -eq 0 ]]; do echo "hello" p=$((p-1)) done
```

📌 Logic difference:

Loop	Runs while...
`while`	condition is **true**
`until`	condition is **false**

📌 5️⃣ Reading from a file in a loop

Given file `see.txt` :

```
hello master zaxxer minecraft roblox undertale
```

Example:

```
while read var1 var2 var3; do echo "$var1 | $var2 | $var3" done < see.txt
```

page 415-433

🛰️ [FIELD DOSSIER] — Bash Debugging, Directory Safety & Case Handling

🎯 ONE-LINE SUMMARY

Use `trouble` or `bash -x` for syntax/debug checks; verify directories exist before operations (`-d`, `cd`); prefer `rm ./*` for safe deletion; `case` patterns match first true condition, use `|` for OR and `;;&` to fall through.

🧠 SECTION A — Syntax Check & Troubleshooting

- **`trouble script_name`** → scans script for syntax errors without execution.
- **Use case:** validate a newly written Bash script before running dangerous commands.

🎯 SECTION B — Safe Directory Checks

Before performing operations inside a directory:

```
#!/bin/bash if [[ ! -d "minecraft" ]]; then echo "Error: directory not found" >&2 exit 1 fi if ! cd "minecraft"; then echo "Error: cannot enter directory" >&2 exit 1 fi echo "rm !"
```

- `[[ ! -d "dir" ]]` → ensures directory exists.
- `cd "dir"` inside `if ! ...` → ensures we can actually enter the directory.
- Only executes subsequent commands (`rm !`) if **both checks succeed**.

Tip:

- Use `rm ./*` instead of `rm *` to delete all files in current directory **safely**.
- Beware of aliases: `alias rm='rm -rf /*'` could be catastrophic.

🎯 SECTION C — Bash Tracing (`-x`) & PS4

Two ways to debug script execution:

1. **Run entire script with trace:**

```
bash -x script_name
```

- Prints every command as executed.
- Prefix for each line can be customized:

```
export PS4='LINENO + '
```

→ shows line number of execution.

2. **Enable tracing within script selectively:**

```
#!/bin/bash echo "Before tracing" set -x # turn on tracing command1 command2
set +x # turn off tracing echo "After tracing"
```

- Allows tracing only critical sections.

🎯 SECTION D — `case` Statement Essentials

Basic syntax:

```
case "$var" in 0) echo "hello world" ;; 1) echo "i number one !" ;; *) echo
"noooo" ;; # default/else esac
```

- Executes **first matching pattern** and exits.
- **OR between patterns** → use `|` (not `||`):

```
case "$val" in a|b) echo "Matches a or b" ;; esac
```

- **Fall-through execution** → use `;;&` to continue testing subsequent patterns:

```
var=a case "$var" in a) echo "a is here!" ;;& [[:alpha:]]) echo "a is back!!"
;;& *) echo "nooo!" ;; esac
```

Notes:

- `*` → catch-all/default.
- `+` or `||` → do **not** work in `case`.
- Use `[[:alpha:]]` or other POSIX character classes for pattern matching.

page 433-450

🧠 Organized Summary — Positional Parameters in Bash

📌 1️⃣ What Are Positional Parameters?

Positional parameters are the arguments passed to a script from the command line.
They are prepared **before the script begins execution**, making them more powerful and consistent than `read`.

Core Variables

Parameter	Meaning
`$0`	Script name / path
`$1`	First argument
`$2`	Second argument
...	...
`$#`	Number of arguments
`$@`	All args (each preserved individually)
`$*`	All args combined into one string

📌 2️⃣ The “9 Arguments Limit” Misunderstanding

Bash **does NOT** limit positional parameters to 9.
The real issue is that:

`$10`

is interpreted as:

`$1 + "0"`

The fix is simply:

`${10} ${11} ${25}`

So Bash supports unlimited parameters — you just need braces for numbers ≥ 10.

📌 3️⃣ `shift` — Processing Many Arguments

`shift` removes `$1` and shifts all remaining parameters left:

Before:

```
$1 = A $2 = B $3 = C $# = 3
```

After `shift`:

```
$1 = B $2 = C $# = 2
```

Typical use-case:

```
while [[ $# -gt 0 ]]; do echo "$1" shift done
```

This lets you process unlimited arguments in a loop.

📌 4️⃣ `$*` vs `$@` — The Critical Difference

Without quotes:

Expression	Behavior
`$*`	Joins all arguments into one string
`$@`	Keeps each argument separate

With quotes (THIS is the important part):

Expression	Behavior
`"$*"`	Treats all arguments as **one string**
`"$@"`	Each argument remains **its own separate element**

> **Always prefer `"$@"` when passing args to another function or command.**

📌 5️⃣ `for` Loops with Positional Parameters

Short form:

```
for x; do
```

This is 100% equivalent to:

```
for x in "$@"; do
```

Other forms:

(1) Range:

```
for i in {1..5}; do
```

(2) C-style:

```
for (( i=0; i<10; i++ )); do
```

📌 6️⃣ `strings` Command in This Context

`strings` extracts readable text from a file — even binary files.

Example:

```
strings /bin/bash
```

It outputs **each word separately**, which is ideal for:

- scanning files
- searching for the longest string
- reading text from binaries

📌 7️⃣ Script: Find the Longest Word in Each File

```
while [[ $# -gt 0 ]]; do file="$1" if [[ -r "$file" ]]; then max_word="" max_len=0 for w in $(strings "$file"); do len=$(echo -n "$w" | wc -c) if (( len > max_len )); then max_len="$len" max_word="$w" fi done echo "$file: '$max_word' ($max_len characters)" fi shift done
```

Why it works?

1. `strings` → extract readable words
2. loop through each word
3. count length using `wc -c`
4. track longest
5. `shift` → move to next file

📌 8️⃣ Exercises

✔️ Exercise 1

Modify the script to also print the **total number of words** extracted from each file.

✔️ Exercise 2

Rewrite the whole script **without using `shift`**, using:

```
for file in "$@"; do
```

✔️ Exercise 3

Add an option `-v` (`--verbose`) that prints progress details.

page 450-470

🧩 Bash Parameter Expansion, Arithmetic & `bc` (Deep Summary)

🧠 Parameter Expansion

Assume:

```
var=12 empty="" unset_var # does not exist
```

🔹 Default Values & Assignment

Syntax	Behavior
`${var:-text}`	Prints `var` if it exists and is non-empty; otherwise prints `text`
`${var:=text}`	If `var` is unset or empty → **assigns** `text` to it
`${var-text}`	Prints `text` **only if** `var` **is unset**
`${var:?error}`	If `var` is unset or empty → **error and exit**
`${var:+text}`	If `var` exists and is non-empty → prints `text` only

⚠️ **Note:** `${var:?message}` does **not** print `message` if the variable exists; it only triggers an error when empty/unset.

🔁 Indirect Expansion & Length

Syntax	Meaning
`${!prefix@}` or `${!prefix*}`	All variable names starting with `prefix`
`${#var}`	Number of characters in the variable

✂️ Substring Expansion

`var="hello world"`

Syntax	Result
`${var:6}`	`world`
`${var:6:2}`	`wo`

🧹 Pattern Removal (`# ## % %%`)

🔹 From the beginning

`a="m.a.c.v"`

Syntax	Result
`${a#*.}`	`a.c.v` (shortest match)
`${a##*.}`	`v` (longest match)

🔹 From the end

`a="minecraft.tar.zip.nmap"`

Syntax	Result
`${a%.*}`	`minecraft.tar.zip`
`${a%%.*}`	`minecraft`

🔄 String Replacement

`a="minecraft.roblox.minecraft"`

Syntax	Behavior
`${a/minecraft/MINE}`	Replace first match only

Syntax	Behavior
`${a//minecraft/MINE}`	Replace all matches
`${a/#minecraft/MINE}`	Replace only at start
`${a/%minecraft/MINE}`	Replace only at end

🧱 Arithmetic Expansion `(())`

`echo $((32)) echo $((0xFF)) # hex → decimal echo $((077)) # octal → decimal echo $((2#1010)) # base#number`

🔹 Ternary Operator (Shortcut if)

`(( a > 100 ? a+=200 : a=0 ))`

⚙️ Bitwise Operators

Operator	Description
`~a`	Bitwise NOT
`a << n`	Left shift (×2ⁿ)
`a >> n`	Right shift (÷2ⁿ)
`a & b`	AND
`a	b`	OR
`a ^ b`	XOR

Examples

`$((5 & 3)) # 1 $((5 | 3)) # 7 $((5 ^ 3)) # 6 $((~5)) # -6`

🧱 bc — Basic Calculator

🔹 Run a file

```
bc file.txt
```

(e.g., file containing `2+2`)

🔹 Interactive mode

```
bc -q 2+2 quit
```

🔹 With Bash (floating-point calculations)

```
echo "scale=2; 5/3" | bc
```

📌 Why use `bc`?

- Supports floating-point arithmetic
- More precise than `$(( ))`
- Useful for real mathematical calculations

page 470-500

🛰️ [FIELD DOSSIER] — Bash Arrays, Exotica, Pipes & FIFO

🎯 ONE-LINE SUMMARY

Indexed arrays store values using numeric indices and often require initialization; quoting determines how array elements are expanded during iteration; Bash provides built-in tools to inspect and modify arrays; pipes (`|`) connect processes directly, while FIFO enables inter-process communication through a named file.

🧠 SECTION A — Indexed Arrays Basics

`a[1]=hello echo "${a[1]}" # hello`

- `declare -a a` → declares an indexed (numeric) array.
- Indices are integers (`0` , `1` , `2` , …).
- Values are accessed directly using the index.

🎯 SECTION B — Preparing & Updating Arrays

Array initialization (fixed-size example):

`for i in {0..23}; do minecraft[i]=0 done`

- Initialization is used when the array size is **known in advance**.
- Improves clarity and avoids ambiguous behavior.

Numeric increment:

`((++minecraft[1])) echo "${minecraft[1]}" # 1`

- `((++array[index]))` increments the numeric value.
- If the element does not exist, Bash treats it as `0` before incrementing.

🎯 SECTION C — Iterating Over Arrays (Quoting Matters)

`animals=("a cat" "a dog" "a cow")`

Without quotes:

`for i in ${animals[@]}; do echo $i; done for i in ${animals[*]}; do echo $i; done`

- Elements are **word-split using IFS**
- `"a cat"` becomes two separate words

With quotes:

`for i in "${animals[@]}"; do echo $i; done`

- Each array element is preserved as a single unit (recommended behavior)

`for i in "${animals[*]}"; do echo $i; done`

- All elements are expanded as **one single string**

🎯 SECTION D — Array Inspection & Manipulation

`echo ${#arrays[@]} # number of elements in the array echo ${#arrays[2]} # length of element at index 2 echo ${!arrays[@]} # list of used indices`

Adding and removing elements:

`arrays+=(pop hi lol) # append elements unset arrays # remove the array entirely`

🎯 SECTION E — Pipes vs FIFO (Exotica)

🔹 Pipe

`command1 | command2`

- Temporary, anonymous data channel
- Exists only during command execution
- Cannot be shared with other processes

🔹 FIFO (Named Pipe)

`mkfifo filename`

Terminal 1:

`cat < filename`

- Waits for incoming data

Terminal 2:

`echo "hello world" > filename`

- Data is written into the FIFO
- Read immediately by the process waiting on the other end

📌 FIFO characteristics:

- Exists as a real filesystem object
- Used for inter-process communication (IPC)
- Writer blocks until a reader exists (and vice versa)

🧠 FINAL NOTE

This chapter bridges:

- **Data structures (arrays)**
- with **process communication mechanisms (pipes and FIFO)**

It represents a shift from executing commands to **system-level thinking**