MATHEMATICAL MODELING FOR COMPUTATIONAL SCIENCE

AM215 - LECTURE 1: Linux and Shell Scripting

Friday, September 5th, 2025

Chris Gumb

Housekeeping

- **PSet 0 is due Tuesday 9/9 @ 11:59PM** (Submit through Gradescope on Canvas)
- **Section Preferences** due by end of today!
- **Sections & OHs** begin next week (See Google Calendar on Canvas)
- **The Ed forum** is the best place to ask questions
- Log into **https://code.harvard.edu** (*https://code.harvard.edu*) to generate an account (more on this later)

Lecture Outline

Unix History

- GNU Project & Linux Kernel

Working in the Linux Terminal

The Basics

- The User (`whoami`)
- Exploring the File System (`~/`, `.`, `..`, `pwd`, `cd`, `ls`, `find`)
- Environment Variables (`$0`, `$HOME`, `$PATH`)
- Aliases

Working with Files

- Creating, Moving, and Deleting Files (`touch`, `mv`, `cp`, `rm`)
- Viewing Files (`cat`, `less`, `head`, `tail`)
- Editing Files (`nano`, `vim`)
- Shell Configuration (`~/.bashrc`, `~/.bash_profile`, `source`)
- File Attributes, Permissions, & Groups (`chown`, `chmod`, `usermod`)

Working with Streams

- Redirection & Pipes (`>`, `|`)
- Regular Expressions & `grep`
- Advanced String Manipulation (`sed`, `awk`)

Shell Scripting

- Command Line Arguments & Functions
- Conditional Statements, Boolean Operators, and Loops
- What is POSIX?

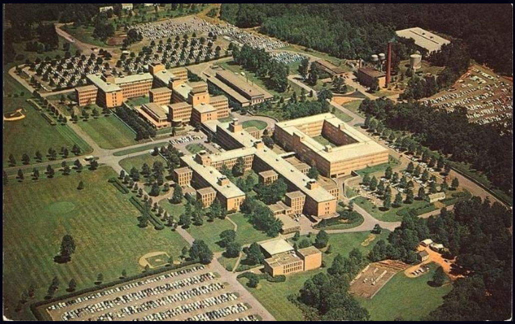
Everyday Skills

- Controlling Processes (`ps`, `top`, `^C`, `kill`)
- Managing Disk Space (`du`)
- Backgrounding & Multiplexing (`^Z`, `fg`, `screen`)

Misc.

- Other Shells
- CLI vs TUI vs GUI

🟦 Unix History: Where It All Began



🟩 In the late 1960s, at Bell Telephone Laboratories in New Jersey, something revolutionary was brewing...

The Birth of Unix

Ken Thompson and **Dennis Ritchie** created Unix in 1969-1970

Why? They wanted a **time-sharing** operating system where:

- Multiple users could share the same computer simultaneously
- Each user could run programs independently
- Resources (CPU, memory, storage) were managed efficiently

Before this, computers typically served only one user at a time!

How Did Users Interact with Unix?

Terminals: Physical devices with keyboards and either:

- Paper printouts (teletypes)
- Cathode ray tube displays (later)

The Shell: A command interpreter that:

- Reads your typed commands
- Executes programs
- Returns results to your terminal

Utility Programs: Small, focused tools that do one job well

- `ls` (list files), `cat` (display file contents), `grep` (search text)
- The Unix philosophy: "Do one thing and do it well"

🟦 But How Do Programs Actually *Do* Anything?

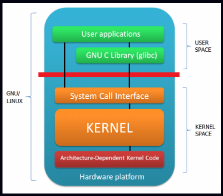
▓▓ When you run a program, it needs to:

- Read keyboard input
- Write to memory and disk
- Display output on screen
- Access hardware devices
- Communicate over networks

▓▓ **The Problem**: Direct hardware access would be chaos with multiple users!

▓▓ *How do we safely coordinate all these requests?*

🟦 The Kernel: Where Mind Meets Metal



The Unix Kernel acts as a gatekeeper:

Kernel Space: Where the operating system runs with full hardware access **User Space**: Where your programs run with restricted permissions

System Calls: The controlled interface between user programs and kernel

- `open()` to access files
- `write()` to output data
- `fork()` to create new processes

This design keeps the system stable and secure!

🟦 The Problem: Unix Wasn't Free

▓▓ By the 1980s, Unix had become:

- **Proprietary** - owned by AT&T
- **Expensive** - required costly licenses
- **Fragmented** - many incompatible versions

▓▓ Universities and researchers were frustrated:

- Couldn't study the source code
- Couldn't modify it for their needs
- Couldn't share improvements

▓▓ *There had to be a better way...*

🟦 Enter Richard Stallman and the GNU Project

1983: Richard Stallman launches the **GNU Project**

- **Goal**: Create a completely free Unix-like system
- **GNU**: "GNU's Not Unix" (computer nerds love recursive acronyms)
- **Philosophy**: Software should be free to use, study, modify, and share

What GNU Accomplished:

- `gcc` (compiler), `bash` (shell), `emacs` (editor)
- Most of the essential Unix utilities
- The GNU General Public License (GPL)

What Was Missing: A kernel! 😱

Linux: The Missing Piece

1991: Finnish student **Linus Torvalds** announces:

> "I'm doing a (free) operating system (just a hobby, won't be big and professional like gnu) for 386(486) AT clones..."

Linux provided:

- A free Unix-like kernel
- The missing piece for GNU's vision
- Collaborative development model

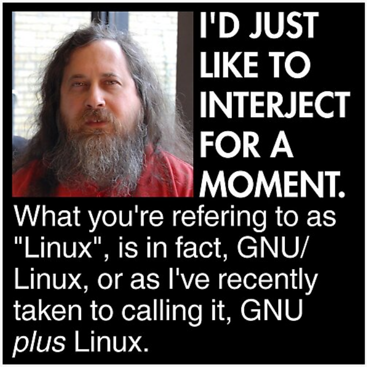
Finally, a complete free Unix-like system was possible!

GNU/Linux: The Complete System

GNU/Linux combines:

- **GNU**: The utilities, compiler, shell, and philosophy
- **Linux**: The kernel that manages hardware

Richard Stallman insists we call it "GNU/Linux" to recognize GNU's contributions!



🟦 What Makes a Linux Distribution?

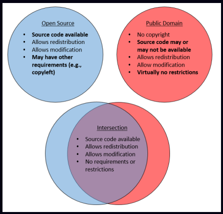
▩ A **Linux Distribution** packages together:

- **Linux kernel** (hardware management)
- **GNU utilities** (basic commands and tools)
- **Package manager** (software installation)
- **Desktop environment** (graphical interface)
- **Default applications** (web browser, text editor, etc.)

▩ **Popular Distributions**:

- **Ubuntu**: User-friendly, great for beginners
- **Debian**: Rock-solid stability, completely free
- **Arch**: Minimalist, cutting-edge, for advanced users
- **Gentoo**: Compile everything from source
- **NixOS**: Reproducible system configurations

Free Software vs Open Source Software



Free Software (GNU/FSF philosophy):

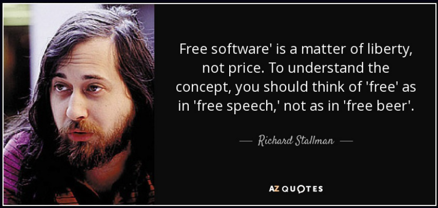
- Emphasizes user **freedom** and **ethics**
- Four essential freedoms: use, study, modify, distribute

Open Source Software:

- Emphasizes **practical benefits**: better quality, security, collaboration
- More business-friendly messaging

Both movements largely support the same software, but with different philosophical emphasis.

We'll revisit software licensing in detail in a later lecture!



🟦 Working in the Linux Terminal

🟩 Now that we understand the history, let's learn to use this powerful system!

What we'll cover:

- Who am I and where am I?
- Navigating the file system
- Understanding files and directories
- Basic file operations
- Environment variables and customization

Time to get our hands dirty!

🟦 First Steps: Who Am I?

When you first open a terminal, you might feel lost. Let's start with the basics:

```
whoami
```

──────────────────── [finished] ────────────────────

```
chris
```

What is a "user" in Linux?

- Every person who uses the system has a unique **username**
- Linux tracks who owns which files and who can access what
- Your username determines your permissions and home directory

Why this matters: Linux was designed as a **multi-user system** from the beginning. Multiple people can use the same computer simultaneously, each with their own files and permissions.

This is different from early personal computers that assumed only one person would use them!

■ Users and Groups

Now let's see more details about your identity:

```
id
```

──────────────────── [finished] ────────────────────

```
uid=1000(chris) gid=100(users)
groups=100(users),1(wheel),17(audio),57(networkmanager),
67(libvirtd),131(docker)
```

What this shows:

- **uid**: Your unique user ID number
- **gid**: Your primary group ID number
- **groups**: All groups you belong to

What are groups? Collections of users who share certain permissions. For example:

- `sudo` group: Can run administrative commands
- `staff` group: Can access shared project files

We'll return to groups when we discuss file permissions - they're crucial for controlling who can access what!

🟦 Where Am I? Understanding Your Location

Every terminal session has a **current working directory** - think of it as "where you are" in the file system:

```
pwd
```

─────────────────── [finished] ───────────────────

```
/home/chris/teaching/AM215/AM215_Lectures_2025/lec01
```

`pwd`: **P**rint **W**orking **D**irectory - shows your current location

What does this path mean?

- It shows your complete address in the file system
- Usually starts with `/home/username` - your personal space
- Think of it like your mailing address - it tells you exactly where you are

Try this: Open a terminal and run `pwd` to see where you are right now!

🟦 The Linux File System: A Hierarchy

Linux organizes everything in a **tree structure** starting from the root `/`:

```
ls /
```

——————————————— [finished] ———————————————

```
bin
boot
dev
etc
home
lib
lib64
mnt
nix
opt
proc
root
run
srv
sys
tmp
usr
var
```

This shows all the top-level directories in the Linux file system.

Important System Directories

What do these directories contain?

- `/`: The root - top of everything
- `/home`: Contains user directories
- `/usr`: System programs and files
- `/etc`: System configuration files
- `/tmp`: Temporary files
- `/bin`: Essential system binaries (programs)
- `/lib` & `/lib64`: System libraries
- `/var`: Variable data (logs, databases)
- `/opt`: Optional software packages
- `/proc`: Virtual filesystem with system information

Note: Different Linux distributions may have slightly different directory structures. For example, some distributions might have `/snap` for Snap packages, or `/nix` for the Nix package manager (as seen here).

```
ls /home
```

[finished]

```
chris
```

This shows all user directories on the system.

```
ls ~
```

[finished]

```
documents
downloads
iso
mnt
music
nix
notes
pictures
repos
school
```

Navigation: Moving Around

Use `cd` (**c**hange **d**irectory) to move around:

```
pwd
```

—————————————————— [finished] ——————————————————

```
/home/chris/teaching/AM215/AM215_Lectures_2025/lec01
```

Let's see where we are first.

```
cd my_project
pwd
```

Try this in a new terminal: Open a new terminal window and run these commands to practice navigation. (Note: `cd` commands don't work in presentation code blocks)

Special directory symbols:

- `.` = current directory
- `..` = parent directory
- `~` = your home directory
- `/` = root directory

🟦 Exploring: What's Here?

The `ls` command **lists** directory contents:

```
ls
```

[finished]

```
am215_lec01_linux_appendix.md
am215_lec01_linux.md
am215_lec01_linux.pdf
config
Documents
img
logs
my_project
old
practice
sample_data.txt
temp_file.txt
```

Basic listing of files and directories.

```
ls -l
```

[finished]

```
total 1912
-rw-r--r-- 1 chris users   14919 Sep  5 01:58
am215_lec01_linux_appendix.md
-rw-r--r-- 1 chris users   62709 Sep  5 02:06
am215_lec01_linux.md
-rw-r--r-- 1 chris users 1867577 Sep  5 02:23
am215_lec01_linux.pdf
drwxr-xr-x 1 chris users      26 Sep  5 01:08 config
drwxr-xr-x 1 chris users      20 Sep  5 00:49 Documents
drwxr-xr-x 1 chris users     438 Sep  5 01:39 img
drwxr-xr-x 1 chris users      14 Sep  5 01:08 logs
drwxr-xr-x 1 chris users     126 Sep  5 01:25 my_project
drwxr-xr-x 1 chris users      30 Sep  4 21:55 old
drwxr-xr-x 1 chris users       0 Sep  5 01:09 practice
```

Understanding Files vs Directories

In Linux, **everything is either a file or a directory**:

```
ls -l
```

[finished]

```
total 1912
-rw-r--r-- 1 chris users   14919 Sep  5 01:58
am215_lec01_linux_appendix.md
-rw-r--r-- 1 chris users   62709 Sep  5 02:06
am215_lec01_linux.md
-rw-r--r-- 1 chris users 1867577 Sep  5 02:23
am215_lec01_linux.pdf
drwxr-xr-x 1 chris users      26 Sep  5 01:08 config
drwxr-xr-x 1 chris users      20 Sep  5 00:49 Documents
drwxr-xr-x 1 chris users     438 Sep  5 01:39 img
drwxr-xr-x 1 chris users      14 Sep  5 01:08 logs
drwxr-xr-x 1 chris users     126 Sep  5 01:25 my_project
drwxr-xr-x 1 chris users      30 Sep  4 21:55 old
drwxr-xr-x 1 chris users       0 Sep  5 01:09 practice
-rw-r--r-- 1 chris users     378 Sep  5 01:08
sample_data.txt
-rw-r--r-- 1 chris users      19 Sep  5 02:25
temp_file.txt
```

The first character in `ls -l` tells you the type:

- `d` = directory (folder)
- `-` = regular file
- `l` = symbolic link (shortcut)

Key insight: Directories are just special files that contain lists of other files!

🟦 What Type of File Is This?

Sometimes you need to know more about a file than just its name:

```
file my_project
```

──────────────────── [finished] ────────────────────

```
my_project: directory
```

The `file` command examines the contents and tells you what type of file something is.

```
file $(which ls)
```

──────────────────── [finished] ────────────────────

```
/run/current-system/sw/bin/ls: symbolic link to
/nix/store/nj8926sbn01v3m4rz4hygmvp33xsm5ld-coreutils-fu
ll-9.7/bin/ls
```

What's happening here?

- `$(which ls)`: This is a **subshell** - it runs `which ls` first and injects the result into the outer command
- `which ls` finds where the `ls` program is located on your system
- `file` then examines that location

What you might see:

- **Binary executable**: A compiled program that the CPU can run directly
- **Symbolic link**: A shortcut that points to another file (common on NixOS and other modern systems)

Why this matters: Not all files are text! Programs can be binary executables or links to other files. The `file` command helps you understand what you're working with, and command substitution lets you chain commands together dynamically.

Creating Your First Directory

Let's create a workspace to practice in:

```
ls -l my_project
```

[finished]

```
total 20
-rwxr-xr-x 1 chris users 503 Sep  5 01:25
file_processor.sh
-rwxr-xr-x 1 chris users 414 Sep  5 01:25 greet_user.sh
-rwxr-xr-x 1 chris users 159 Sep  5 01:25
hello_script.sh
-rw-r--r-- 1 chris users  78 Sep  5 00:49 README.md
-rwxr-xr-x 1 chris users  46 Sep  5 00:49 script.py
```

`mkdir`: **M**a**k**e **dir**ectory - creates a new folder

Notice the `d` at the beginning of the permissions - this shows it's a directory.

What happened? The `my_project` directory already exists in our lecture folder. You can create new directories with `mkdir dirname`.

Moving Into Our New Directory

Now let's enter the directory we just created:

```
cd my_project
pwd
```

Try this in a new terminal: `cd` changes your location, but it doesn't work in presentation code blocks. Open a new terminal and try these commands.

Think of it like this: If the file system is like a building, `cd` is like walking from one room to another. `pwd` tells you which room you're currently in.

Creating Your First Files

Let's create some empty files to work with:

```
ls -l my_project/
```

[finished]

```
total 20
-rwxr-xr-x 1 chris users 503 Sep  5 01:25
file_processor.sh
-rwxr-xr-x 1 chris users 414 Sep  5 01:25 greet_user.sh
-rwxr-xr-x 1 chris users 159 Sep  5 01:25
hello_script.sh
-rw-r--r-- 1 chris users  78 Sep  5 00:49 README.md
-rwxr-xr-x 1 chris users  46 Sep  5 00:49 script.py
```

`touch`: Creates an empty file (or updates the timestamp if the file already exists)

What do you notice about the files in my_project/?

- The README.md file contains some content
- The first character is `-` showing it's a regular file
- You can create empty files with `touch filename`

Why start with empty files? It's often useful to create placeholder files that you'll fill with content later.

Adding Content to Files

Let's put some content in our file:

```
echo "# My First Project"
```

[finished]

```
# My First Project
```

`echo`: Prints text to the screen

```
echo "# My First Project" > temp_file.txt
```

[finished]

The `>` symbol **redirects** the output into a file instead of showing it on screen.

What just happened? Instead of printing to the screen, we saved the text into a new file called `temp_file.txt`. The `>` is like saying "put this text into that file instead of showing it to me."

Viewing File Contents

Now let's see what's inside our file:

```
cat my_project/README.md
```

[finished]

```
# My First Project

This is a sample project directory for the Linux
lecture.
```

`cat`: Displays the contents of a file (**cat**enate and print)

```
ls -l my_project/README.md
```

[finished]

```
-rw-r--r-- 1 chris users 78 Sep  5 00:49
my_project/README.md
```

Notice the file size - it contains text content!

Why is it called `cat`? Originally it was designed to con**cat**enate (join together) multiple files, but it's commonly used to display single files too.

Understanding Paths: Where Things Live

Every file and directory has an **address** called a path. Let's explore this:

```
pwd
```

[finished]

```
/home/chris/teaching/AM215/AM215_Lectures_2025/lec01
```

This shows our **absolute path** - the complete address from the root of the system.

Two types of paths:

- **Absolute paths**: Start with `/` - work from anywhere (like a complete mailing address)
- **Relative paths**: Start from where you are now (like "go to the kitchen")

Think of it like directions: "123 Main Street" works from anywhere (absolute), but "go upstairs" only makes sense from where you currently are (relative).

Relative Paths in Action

Let's see relative paths in action:

```
ls my_project/README.md
```

[finished]

```
my_project/README.md
```

This uses a **relative path** - we're looking for README.md inside the my_project directory.

```
ls ./my_project/README.md
```

[finished]

```
./my_project/README.md
```

The `./` explicitly means "starting from the current directory" - it's the same as the previous command.

The dot `.` is special: It always means "the current directory" - wherever you happen to be right now.

Going Back: The Parent Directory

Let's explore moving up in the directory tree:

```
ls ..
```

[finished]

```
general_approach.md
lec01
lec02
README.md
```

The `..` means "parent directory" - the directory that contains our current directory.

```
cd ..
pwd
```

Try this in a new terminal: The `cd` command doesn't work in presentation blocks, but you can try it in a new terminal.

The `..` is like an "up" button - it takes you to the directory that contains the one you're in.

🟦 Finding Files: When You Don't Know Where They Are

Sometimes you need to locate files in the system:

```
find . -name "README.md"
```

——————————————————— [finished] ———————————————————

```
./my_project/README.md
```

`find`: Searches for files and directories by name, type, size, and more

What this command means:

- `find`: The search command
- `.`: Start searching from the current directory
- `-name "README.md"`: Look for something with this exact name

Why is this useful? As your projects grow, you might have hundreds of files. `find` helps you locate specific ones quickly.

Getting Help: The Manual System

Linux has built-in documentation for every command:

```
man ls
```

──────────────────── [finished] ────────────────────

```
LS(1)                               User Commands
LS(1)

NAME
       ls - list directory contents

SYNOPSIS
       ls [OPTION]... [FILE]...

DESCRIPTION
       List information about the FILEs (the current
directory by default).  Sort en-
       tries alphabetically if none of -cftuvSUX nor
--sort is specified.

       Mandatory arguments to long options are mandatory
for short options too.

       -a, --all
              do not ignore entries starting with .

       -A, --almost-all
              do not list implied . and ..

       --author
              with -l, print the author of each file

       -b, --escape
              print C-style escapes for nongraphic
characters

       --block-size=SIZE
              with   -l,   scale   sizes   by   SIZE
when   printing   them;  e.g.,
              '--block-size=M'; see SIZE format below

       -B, --ignore-backups
```

Environment Variables: Your Shell's Memory

Your shell remembers information in **environment variables** - think of them as the shell's memory:

```
echo $USER
```

[finished]

```
chris
```

What are environment variables?

- Variables that store information your shell and programs need
- Always written in ALL CAPS by convention
- Referenced with a `$` symbol

Why do we need them? Programs need to know things like:

- Who you are (`$USER`)
- Where your home directory is
- Where to find other programs

Important Environment Variables

Let's explore some key environment variables:

```
echo $HOME
```

──────────────────── [finished] ────────────────────

```
/home/chris
```

This shows your home directory - where your personal files live.

```
echo $SHELL
```

──────────────────── [finished] ────────────────────

```
/run/current-system/sw/bin/zsh
```

This shows which shell program you're using (probably `/bin/bash` or `/bin/zsh`).

These variables help programs work correctly: When a program needs to save a file to your home directory, it uses `$HOME` to know where that is.

🟦 The PATH: How Linux Finds Programs

The most important environment variable is `$PATH`:

```
echo $PATH
```

──────────────── [finished] ────────────────

```
/nix/store/8z64p7f9bzk4csybdcclqvsmhr2mzng2-python3.13-w
easyprint-65.1/bin:/nix/store/4wvmqr6mnhdni0cjcqhyhk5bky
mhvw0d-source/.local/bin:/run/wrappers/bin:/home/chris/.
nix-profile/bin:/nix/profile/bin:/home/chris/.local/stat
e/nix/profile/bin:/etc/profiles/per-user/chris/bin:/nix/
var/nix/profiles/default/bin:/run/current-system/sw/bin:
/nix/store/lv91pnk6dqvw0xmbi5irli7m6nikfr33-binutils-wra
pper-2.44/bin:/nix/store/dsx92353f5qrf3jh7q02zj5mlpm17rd
n-hyprland-qtutils-0.1.4/bin:/nix/store/5h5758kspk2ir12d
vwrgilmcizz617kh-pciutils-3.14.0/bin:/nix/store/43vw43b1
km56y6idkzrbz5narkhlfca4-pkgconf-wrapper-2.4.3/bin:/nix/
store/k75dxigqjiqbg7i9rfkmx4m9q4fpv6h5-kitty-0.42.2/bin:
/nix/store/71zmb5blvs1w6fkiwayidzx8mbmqiyl7-imagemagick-
7.1.2-0/bin:/nix/store/z1l05nn4xyaxv25f9pvi7bkmw6jmb48c-
ncurses-6.5-dev/bin:/home/chris/.config/zsh/plugins/zsh-
you-should-use:/home/chris/.config/zsh/plugins/zsh-vi-mo
de
```

What is PATH?

- A list of directories where the shell looks for programs
- Directories are separated by colons `:`
- Searched in order from left to right

How it works:

1. You type `ls`
2. Shell checks first directory in PATH for `ls`
3. If not found, checks next directory
4. Continues until found (or gives up)

Seeing PATH in Action

Let's see where common programs live:

```
which ls
```

[finished]

```
/run/current-system/sw/bin/ls
```

`which`: Shows the full path to a program

```
which python3
```

[finished]

```
/home/chris/.nix-profile/bin/python3
```

This shows where Python is installed on your system.

Why `./script.py` needs the `./`: The current directory (`.`) is usually NOT in your PATH for security reasons. So you must explicitly tell the shell to look in the current directory with `./`.

🟦 Creating Your Own Environment Variables

You can create your own environment variables:

```bash
MY_NAME="Student"
echo $MY_NAME
```

Try this in a new terminal: Variable assignments don't persist in presentation code blocks. Open a new terminal to try setting and using variables.

Set a variable (no spaces around `=`!) and then use it with `$`

Important: This variable only exists in your current shell session. When you close the terminal, it's gone. Later we'll learn how to make variables permanent.

Aliases: Shortcuts for Common Commands

Aliases let you create shortcuts for commands you use frequently:

```
alias ll='ls -la'
ll
```

Try this in a new terminal: Aliases don't persist in presentation code blocks. Open a new terminal to try creating and using aliases.

Create an alias called `ll` that runs `ls -la`, then use it!

Why use aliases?

- Save typing for long commands
- Remember complex options more easily
- Customize commands to your preferences

🟦 Useful Alias Examples

Here are some popular aliases that make life easier:

```
alias la='ls -la'
alias welcome='cowsay "Hello $USER, welcome to Linux!"'
welcome
```

Try this in a new terminal: Aliases don't work in presentation code blocks. Open a new terminal to try these commands.

Create useful aliases and a fun welcome message! (Note: `cowsay` might not be installed on all systems)

Try creating your own: Think about commands you type often and create shortcuts for them!

🟦 Viewing Your Current Aliases

See what aliases you have defined:

```
alias
unalias welcome
```

Try this in a new terminal: Alias commands don't work in presentation code blocks. Open a new terminal to try viewing and removing aliases.

Use `alias` to see all current aliases, and `unalias name` to remove specific ones.

Remember: Like variables, aliases only last for your current session unless you make them permanent.

🟦 Working with Files: Basic Operations

Now let's learn to manipulate files and directories:

```
cp my_project/README.md my_project/README_backup.md
ls my_project/
```

Try this in a new terminal: File operations change the system state, so try these commands in a separate terminal.

Essential file operations:

- `cp source dest`: **Cop**y files or directories
- `mv source dest`: **Move** (rename) files or directories
- `rm filename`: **Re**move files (**Be careful! No trash bin!**)
- `rmdir dirname`: Remove empty directories

Why practice copying first? It's safer to copy before you move or delete - you always have a backup!

🟦 Viewing File Contents: Beyond `cat`

Different ways to examine files:

```
cat sample_data.txt
```

[finished]

```
Line 1: This is the first line
Line 2: Here's some sample data
Line 3: More content for demonstration
Line 4: We can use this for head/tail examples
Line 5: This file has multiple lines
Line 6: Perfect for learning file commands
Line 7: Each line is numbered for clarity
Line 8: This helps with understanding output
Line 9: Almost at the end now
Line 10: This is the final line
```

`cat`: Shows the entire file content

```
head -3 sample_data.txt
```

[finished]

```
Line 1: This is the first line
Line 2: Here's some sample data
Line 3: More content for demonstration
```

`head -n`: Shows the first n lines (great for previewing large files)

```
tail -3 sample_data.txt
```

[finished]

```
Line 8: This helps with understanding output
Line 9: Almost at the end now
```

🟦 Working with Larger Files: `less` is More

For large files, use `less` (a pager):

```
less sample_data.txt
```

Try this in a new terminal: `less` is interactive and doesn't work in presentation blocks.

`less` **navigation:**

- `Space` or `f`: Forward one page
- `b`: Back one page
- `j`/`k`: Down/up one line
- `/pattern`: Search forward
- `?pattern`: Search backward
- `q`: Quit

Why `less`? It doesn't load the entire file into memory - perfect for huge log files that might be gigabytes in size!

Counting and Measuring Files

Let's learn about file statistics:

```
wc sample_data.txt
```

[finished]

```
 10  71 378 sample_data.txt
```

`wc`: Counts lines, words, and characters

What the numbers mean:

- First number: lines
- Second number: words
- Third number: characters (bytes)

```
wc -l sample_data.txt
```

[finished]

```
10 sample_data.txt
```

`wc -l`: Count only lines (very useful for data analysis)

Real-world use: "How many entries are in this dataset?" → `wc -l data.csv`

🟦 File Operations: Moving and Copying

Let's practice safe file management:

```bash
# First, let's see what we have
ls my_project/

# Copy a file (safe - creates duplicate)
cp my_project/README.md my_project/README_backup.md

# Move/rename a file
mv my_project/README_backup.md my_project/BACKUP.md

# Check our work
ls my_project/
```

Try this in a new terminal: These commands modify files, so practice in a separate terminal.

Safety tip: Always `ls` before and after file operations to confirm what happened!

The difference:

- `cp`: Creates a copy, original remains
- `mv`: Moves/renames, original is gone

Creating Directory Structures

Organize your files with directories:

```
# Create a single directory
mkdir practice

# Create nested directories at once
mkdir -p practice/data/raw
mkdir -p practice/data/processed
mkdir -p practice/scripts

# See the structure
ls -la practice/
ls -la practice/data/
```

Try this in a new terminal: Directory creation changes the filesystem.

The `-p` flag: Creates parent directories as needed - very handy for deep directory structures!

Why organize? As projects grow, good organization saves hours of searching for files.

🟦 Removing Files and Directories

⚠️ DANGER ZONE: No Undo!

```
# Remove a file (PERMANENT!)
rm practice/temp_file.txt

# Remove an empty directory
rmdir practice/empty_dir

# Remove directory and all contents (VERY DANGEROUS!)
rm -r practice/old_project

# Remove with confirmation prompts (safer)
rm -i important_file.txt
```

Try this in a new terminal: Practice with test files first!

Critical safety tips:

- Linux has **no trash bin** by default
- `rm` is permanent - files are gone forever
- Always double-check your command before pressing Enter
- Use `rm -i` for interactive confirmation
- Never run `rm -rf /` (this would delete everything!)

File Permissions Deep Dive

Understanding who can access your files:

```
ls -l my_project/README.md
```

──────────────────── [finished] ────────────────────

```
-rw-r--r-- 1 chris users 78 Sep  5 00:49
my_project/README.md
```

Let's decode this: `-rw-r--r--`

Permission structure: `(type)(owner)(group)(other)`

- First character: file type (`-` = file, `d` = directory)
- Next 3: owner permissions (you)
- Next 3: group permissions (your group)
- Last 3: other permissions (everyone else)

Permission types:

- `r`: read (view contents)
- `w`: write (modify contents)
- `x`: execute (run as program)

Changing File Permissions

Control who can access your files:

```
# Make a file executable
chmod +x my_project/script.py

# Remove write permission for group and others
chmod go-w my_project/README.md

# Set specific permissions with numbers
chmod 755 my_project/script.py  # rwxr-xr-x
chmod 644 my_project/README.md  # rw-r--r--

# Check the results
ls -l my_project/
```

Try this in a new terminal: Permission changes affect the filesystem.

Common permission patterns:

- `644`: Files (owner read/write, others read-only)
- `755`: Executables (owner all, others read/execute)
- `700`: Private files (owner only)

Octal notation: `4`=read, `2`=write, `1`=execute. Add them up!

Streams and Redirection: Controlling Data Flow

Every program has three streams:

```
# Redirect output to a file
echo "Hello World" > output.txt

# Append to a file (don't overwrite)
echo "Second line" >> output.txt

# Redirect errors separately
ls nonexistent_file 2> errors.txt

# Redirect both output and errors
ls my_project/ nonexistent_file &> combined.txt
```

Try this in a new terminal: Redirection creates/modifies files.

Stream types:

- **stdin** (0): Input to programs
- **stdout** (1): Normal output
- **stderr** (2): Error messages

Redirection operators:

- `>`: Redirect stdout (overwrites)
- `>>`: Append stdout
- `2>`: Redirect stderr
- `&>`: Redirect both stdout and stderr

The Power of Pipes: Connecting Commands

Pipes let you chain commands together:

```
cat sample_data.txt | head -5
```

[finished]

```
Line 1: This is the first line
Line 2: Here's some sample data
Line 3: More content for demonstration
Line 4: We can use this for head/tail examples
Line 5: This file has multiple lines
```

This shows the first 5 lines of the file.

```
ls -la | wc -l
```

[finished]

```
15
```

This counts how many files are in the current directory.

The Unix Philosophy: Small tools that do one thing well, connected by pipes

Common pipe patterns:

- `command | head -n`: Limit output to first n lines
- `command | tail -n`: Show last n lines
- `command | wc -l`: Count lines of output
- `command | sort`: Sort the output alphabetically

Practical File Management Example

Let's put it all together with a real scenario:

```
# Create a project structure
mkdir -p project/{src,docs,tests,data}

# Copy some files to organize
cp sample_data.txt project/data/
cp my_project/README.md project/docs/

# Check our work
ls -la project/
ls -la project/data/

# Create a summary
echo "Project created on $(date)" > project/README.md
echo "Data files: $(ls project/data/ | wc -l)" >>
project/README.md

# View our summary
cat project/README.md
```

Try this in a new terminal: This creates a complete project structure.

This demonstrates: directory creation, file copying, redirection, command substitution, and pipes working together!

Editing Files: Your First Text Editor

Sometimes you need to create or modify text files. Let's start with `nano` - a beginner-friendly editor:

```
nano my_first_file.txt
```

Try this in a new terminal: Text editors are interactive programs that don't work in presentation blocks.

Basic `nano` commands:

- Type normally to add text
- `Ctrl+O`: Save (write **O**ut)
- `Ctrl+X`: Exit
- `Ctrl+K`: Cut entire line
- `Ctrl+U`: Paste (uncut)
- `Ctrl+W`: Search (where is)

Why learn a terminal editor? Sometimes you're working on remote servers with no graphical interface - terminal editors are your only option!

Advanced Editing with Vim

`vim` is a more powerful but complex editor:

```
vim practice_file.txt
```

Try this in a new terminal: Vim has a steep learning curve but is incredibly powerful.

Vim has different modes:

- **Normal mode**: Navigate and run commands (default)
- **Insert mode**: Type text (press `i` to enter)
- **Command mode**: Save, quit, search (press `:` to enter)

Essential vim commands:

- `i`: Enter insert mode
- `Esc`: Return to normal mode
- `:w`: Save file
- `:q`: Quit
- `:wq`: Save and quit
- `:q!`: Quit without saving

Warning: Vim can be confusing at first - many people get "trapped" in it! Always remember `Esc` then `:q!` to escape.

Which Editor Should You Use?

For beginners: Start with `nano`

- Simple and intuitive
- Shows commands at the bottom
- Works like most modern editors

For power users: Learn `vim` (or `emacs`)

- Extremely efficient once mastered
- Available on virtually every Unix system
- Powerful features for programming

Modern alternatives:

- `code` (VS Code) - if available
- `micro` - nano-like but more modern
- `helix` - modern vim-inspired editor

Try this: Create a simple text file with your chosen editor and practice the basic commands.

Shell Configuration: Making Changes Permanent

Remember those aliases and variables that disappear when you close the terminal? Let's fix that:

```
echo $HOME
ls -la ~ | grep bash
```

Try this in a new terminal: Look for configuration files in your home directory.

Key configuration files:

- `~/.bashrc`: Bash configuration (most common)
- `~/.bash_profile`: Login shell configuration
- `~/.zshrc`: Zsh configuration (macOS default)

What goes in these files?

- Environment variables
- Aliases
- Custom functions
- PATH modifications

Editing Your Shell Configuration

Let's customize your shell by editing `~/.bashrc`:

```
# Back up your current config first
cp ~/.bashrc ~/.bashrc.backup

# Edit your configuration
nano ~/.bashrc

# Add these lines to the end:
# Custom aliases
alias ll='ls -la'
alias la='ls -A'
alias l='ls -CF'

# Custom environment variables
export EDITOR=nano
export HISTSIZE=10000

# Custom PATH (if needed)
export PATH="$HOME/bin:$PATH"
```

Try this in a new terminal: Always backup configuration files before editing them!

After editing: You need to reload the configuration or start a new terminal session.

Reloading Your Configuration

After editing your shell configuration, you need to apply the changes:

```
source ~/.bashrc
```

Try this in a new terminal: The `source` command reloads your configuration.

Alternative methods:

- `. ~/.bashrc` (dot is shorthand for source)
- Close and reopen your terminal
- Start a new terminal session

Test your changes:

```
# Try your new aliases
ll
la

# Check your environment variables
echo $EDITOR
echo $HISTSIZE
```

Pro tip: Always test your configuration changes in a new terminal to make sure they work!

Common Shell Customizations

Here are some popular customizations to add to your `~/.bashrc`:

```bash
# Better history management
export HISTCONTROL=ignoredups:erasedups
export HISTSIZE=10000
export HISTFILESIZE=20000

# Colorful output
alias ls='ls --color=auto'
alias grep='grep --color=auto'

# Safety aliases
alias rm='rm -i'
alias cp='cp -i'
alias mv='mv -i'

# Useful shortcuts
alias ..='cd ..'
alias ...='cd ../..'
alias h='history'
alias c='clear'

# Show current directory in terminal title
export PS1='\[\e]0;\w\a\]\u@\h:\w\$ '
```

Try this in a new terminal: Add these customizations gradually and test each one.

Remember: After adding these, run `source ~/.bashrc` to apply them immediately.

Understanding Login vs Non-Login Shells

Different shell types read different configuration files:

Login shell (when you first log in):

1. `/etc/profile` (system-wide)
2. `~/.bash_profile` (if it exists)
3. `~/.bash_login` (if no .bash_profile)
4. `~/.profile` (if neither above exists)

Non-login shell (new terminal windows):

- `~/.bashrc`

The solution: Most people put this in `~/.bash_profile`:

```
# Source .bashrc if it exists
if [ -f ~/.bashrc ]; then
    source ~/.bashrc
fi
```

This ensures: Your customizations work in both login and non-login shells.

Best practices:

- Always backup before editing (`cp ~/.bashrc ~/.bashrc.backup`)
- Test changes in a new terminal
- Add changes gradually

For detailed troubleshooting steps, see the appendix slides.

🟦 Text Processing: `grep` for Pattern Matching

`grep` is essential for searching text:

```
cat sample_data.txt | grep "Line"
```

———————————————— [finished] ————————————————

```
Line 1: This is the first line
Line 2: Here's some sample data
Line 3: More content for demonstration
Line 4: We can use this for head/tail examples
Line 5: This file has multiple lines
Line 6: Perfect for learning file commands
Line 7: Each line is numbered for clarity
Line 8: This helps with understanding output
Line 9: Almost at the end now
Line 10: This is the final line
```

`grep`: **G**lobally search for a **R**egular **E**xpression and **P**rint matching lines

```
grep -n "data" sample_data.txt
```

———————————————— [finished] ————————————————

```
2:Line 2: Here's some sample data
```

Useful `grep` options:

- `-i`: Case insensitive
- `-n`: Show line numbers
- `-r`: Recursive (search directories)
- `-v`: Invert match (show non-matching lines)
- `-c`: Count matches

Why is grep powerful? It can search through thousands of files in seconds, making it essential for code analysis and log investigation.

🟦 Regular Expressions: Pattern Power

`grep` supports powerful pattern matching with **regular expressions**:

```
echo -e "cat\ndog\nbat\nrat" | grep "at"
```

──────────────────── [finished] ────────────────────

```
cat
bat
rat
```

This matches any line containing "at"

```
echo -e "cat\ndog\nbat\nrat" | grep "^.at$"
```

──────────────────── [finished] ────────────────────

```
cat
bat
rat
```

This matches lines that start with any character, followed by "at", and end there.

Basic regex patterns:

- `.`: Any single character
- `^`: Start of line
- `$`: End of line
- `*`: Zero or more of previous character
- `[abc]`: Any of a, b, or c
- `[A-Z]`: Any uppercase letter

Regular Expressions: Advanced Patterns

Let's explore more complex patterns:

```
grep "^Line [0-9]" sample_data.txt
```

[finished]

```
Line 1: This is the first line
Line 2: Here's some sample data
Line 3: More content for demonstration
Line 4: We can use this for head/tail examples
Line 5: This file has multiple lines
Line 6: Perfect for learning file commands
Line 7: Each line is numbered for clarity
Line 8: This helps with understanding output
Line 9: Almost at the end now
Line 10: This is the final line
```

This finds lines starting with "Line" followed by a space and a digit.

```
grep -E "Line [1-3]:" sample_data.txt
```

[finished]

```
Line 1: This is the first line
Line 2: Here's some sample data
Line 3: More content for demonstration
```

Extended regex patterns (use `-E` flag):

- `+`: One or more of previous
- `?`: Zero or one of previous
- `{n,m}`: Between n and m occurrences
- `|`: OR operator
- `()`: Grouping

Real-world example: Finding email addresses, phone numbers, or specific log patterns in large files.

Practical Grep Examples

Let's see grep in action with real scenarios:

```
grep -i "error" logs/app.log
```

[finished]

```
2024-09-05 10:30:19 ERROR Failed to process
request
```

Find all error messages (case insensitive)

```
grep -c "INFO" logs/app.log
```

[finished]

```
3
```

Count how many INFO messages we have

```
grep -v "DEBUG" logs/app.log
```

[finished]

```
2024-09-05 10:30:15 INFO Application started
2024-09-05 10:30:17 INFO Database connection
established
2024-09-05 10:30:18 WARN Deprecated API endpoint
used
2024-09-05 10:30:19 ERROR Failed to process request
2024-09-05 10:30:20 INFO Request processed
successfully
```

Show all lines except DEBUG messages

sed: The Stream Editor

`sed` is a powerful **stream editor** for filtering and transforming text:

```
echo "Hello World" | sed 's/World/Linux/'
```

[finished]

```
Hello Linux
```

What is sed?

- Created in 1973 by Lee E. McMahon at Bell Labs
- **S**tream **Ed**itor - processes text line by line
- Perfect for automated text transformations
- Works with pipes and files

Basic sed syntax: `sed 's/pattern/replacement/flags'`

- `s`: substitute command
- `/`: delimiter (can use other characters)
- `g`: global flag (replace all occurrences on each line)

Why use sed? When you need to make the same change to many files or many lines - sed can do it instantly.

🟦 sed in Action: Real Examples

Let's see sed's power with practical examples:

```
sed 's/Line/Row/g' sample_data.txt
```

[finished]

```
Row 1: This is the first line
Row 2: Here's some sample data
Row 3: More content for demonstration
Row 4: We can use this for head/tail examples
Row 5: This file has multiple lines
Row 6: Perfect for learning file commands
Row 7: Each line is numbered for clarity
Row 8: This helps with understanding output
Row 9: Almost at the end now
Row 10: This is the final line
```

Replace all occurrences of "Line" with "Row"

```
sed -n '2,4p' sample_data.txt
```

[finished]

```
Line 2: Here's some sample data
Line 3: More content for demonstration
Line 4: We can use this for head/tail examples
```

Print only lines 2 through 4 (`-n` suppresses default output, `p` prints)

More sed capabilities:

- Delete lines: `sed '3d'` (delete line 3)
- Insert text: `sed '2i\New line'` (insert before line 2)
- Multiple commands: `sed 's/old/new/g; s/foo/bar/g'`

Real-world use: Configuration file updates, log processing, code refactoring across multiple files.

awk: The Pattern-Action Language

`awk` is a complete programming language for text processing:

```
echo -e "apple 5\nbanana 3\ncherry 8" | awk '{print $1,
$2 * 2}'
```

[finished]

```
apple 10
banana 6
cherry 16
```

What is awk?

- Created in 1977 by Aho, Weinberger, and Kernighan at Bell Labs
- Named after their initials: **A**ho, **W**einberger, **K**ernighan
- A complete programming language, not just a text tool
- Excels at processing structured data (columns, fields)

Basic awk concepts:

- Automatically splits input into fields (`$1`, `$2`, etc.)
- `$0` represents the entire line
- Built-in variables: `NR` (line number), `NF` (number of fields)

Why use awk? When you need to perform calculations, format output, or process columnar data.

awk: Advanced Text Processing

Let's explore awk's programming capabilities:

```
awk 'BEGIN {print "Processing data..."} {sum += $2} END
{print "Total:", sum}' <<< $'item1 10\nitem2 20\nitem3
15'
```

──────────────────────────── [finished] ────────────────────────────

```
Processing data...
Total: 45
```

This calculates the sum of the second column.

awk structure:

- `BEGIN {}`: Executed before processing any input
- `{action}`: Executed for each input line
- `END {}`: Executed after all input is processed

More awk features:

- Conditional processing: `awk '$2 > 10 {print $1}'`
- Built-in functions: `length()`, `substr()`, `gsub()`
- Variables and arrays for complex data processing

Real-world use: Log analysis, CSV processing, generating reports from structured data.

🟦 Putting It All Together: Text Processing Pipeline

Let's combine grep, sed, and awk in a real workflow:

```
cat logs/app.log | grep "INFO" | sed '
s/INFO/INFORMATION/' | awk '{print "Log entry:", $1, $2,
"Message:", $4}'
```

──────────────────────────────── [finished] ────────────────────────────────

```
Log entry: 2024-09-05 10:30:15 Message: Application
Log entry: 2024-09-05 10:30:17 Message: Database
Log entry: 2024-09-05 10:30:20 Message: Request
```

This pipeline:

1. **grep**: Filters for INFO messages
2. **sed**: Changes "INFO" to "INFORMATION"
3. **awk**: Reformats the output with custom labels

The Unix Philosophy in action: Small, focused tools working together to solve complex problems.

When to use each:

- **grep**: Finding and filtering text
- **sed**: Simple find-and-replace operations
- **awk**: Complex data processing and calculations

Pro tip: Start simple with one tool, then add others to the pipeline as needed!

■ Shell Scripting: Automating Your Work

What is shell scripting?

- A way to save and replay sequences of commands
- Automate repetitive tasks
- Combine multiple programs to solve complex problems
- Create your own custom tools

Why learn shell scripting?

- Save time on repetitive tasks
- Ensure consistency in your workflows
- Share your solutions with others
- Essential skill for system administration and data processing

Think of shell scripts as recipes - once written, you can follow them perfectly every time!

Your First Shell Script

Let's create a simple script:

```
cat my_project/hello_script.sh
```

─────────────────── [finished] ───────────────────

```
#!/usr/bin/env bash
# Simple greeting script

echo "Hello from my first shell script!"
echo "Today is $(date)"
echo "You are running this script from:
$(pwd)"
```

What makes this a script?

- **Shebang** (`#!/usr/bin/env bash`): Tells the system which interpreter to use
- **Comments** (`#`): Document what the script does
- **Commands**: Same commands you'd type in the terminal

```
chmod +x my_project/hello_script.sh
./my_project/hello_script.sh
```

Try this in a new terminal: Scripts need execute permission to run.

Why `#!/usr/bin/env bash`? This finds bash wherever it's installed, making your script more portable.

The Shebang: Choosing Your Interpreter

The **shebang** (`#!`) tells the system which program should run your script:

```
#!/usr/bin/env bash    # Bash script
#!/usr/bin/env python3 # Python script
#!/usr/bin/env zsh     # Zsh script
```

Why use `/usr/bin/env`?

- Finds the interpreter in your PATH
- More portable than hardcoding paths like `/bin/bash`
- Respects user customizations

Best practice: Always include a shebang as the very first line of your script.

File extensions: While `.sh` is common, it's optional. The shebang determines how the script runs, not the filename.

Script Arguments: Making Scripts Flexible

Scripts can accept arguments just like regular commands:

```
cat my_project/greet_user.sh
```

[finished]

```
#!/usr/bin/env bash
# Script that uses command line arguments

if [ $# -eq 0 ]; then
    echo "Usage: $0 <name>"
    echo "Please provide your name as an
argument"
    exit 1
fi

NAME=$1
TIME=$(date +%H)

if [ $TIME -lt 12 ]; then
    GREETING="Good morning"
elif [ $TIME -lt 18 ]; then
    GREETING="Good afternoon"
else
    GREETING="Good evening"
fi

echo "$GREETING, $NAME!"
echo "Welcome to shell scripting!"
```

Special variables for arguments:

- `$0`: Script name
- `$1, $2, $3...`: First, second, third arguments
- `$#`: Number of arguments
- `$@`: All arguments as separate words
- `$*`: All arguments as a single string

```
chmod +x my_project/greet_user.sh
./my_project/greet_user.sh Alice
```

Variables in Shell Scripts

Shell scripts use variables to store and manipulate data:

```
# Setting variables (no spaces around =!)
NAME="Alice"
COUNT=42
CURRENT_DIR=$(pwd)

# Using variables (always use $ to reference)
echo "Hello, $NAME"
echo "Count is: $COUNT"
echo "Working in: $CURRENT_DIR"
```

Variable naming rules:

- Use letters, numbers, and underscores
- Start with a letter or underscore
- Convention: UPPERCASE for environment variables, lowercase for local variables

Command substitution: `$(command)` runs the command and uses its output as the variable value.

Quoting in Shell Scripts

Proper quoting is crucial for reliable scripts:

```
# Hard quotes (single): No variable expansion
MESSAGE='Hello $USER'
echo $MESSAGE  # Prints: Hello $USER

# Soft quotes (double): Variables are expanded
MESSAGE="Hello $USER"
echo $MESSAGE  # Prints: Hello chris

# No quotes: Word splitting and globbing occur
FILES=$(ls *.txt)
echo $FILES    # May break with spaces in filenames
echo "$FILES"  # Safer - preserves spaces
```

Best practice: Always quote your variables: `"$VARIABLE"`

When to use each:

- Single quotes: Literal strings
- Double quotes: Strings with variables
- No quotes: Only when you want word splitting

Conditional Statements: Making Decisions

Scripts can make decisions using `if` statements:

```
if [ condition ]; then
    # commands if condition is true
elif [ other_condition ]; then
    # commands if other_condition is true
else
    # commands if all conditions are false
fi
```

String comparisons:

- `[ "$str1" = "$str2" ]`: Equal
- `[ "$str1" != "$str2" ]`: Not equal
- `[ -z "$str" ]`: String is empty
- `[ -n "$str" ]`: String is not empty

Always quote variables in tests! `[ "$var" = "value" ]` not `[ $var = value ]`

Numeric Comparisons

For numbers, use different operators:

```
if [ $# -gt 0 ]; then
    echo "You provided $# arguments"
else
    echo "No arguments provided"
fi
```

Numeric comparison operators:

- `-eq`: Equal
- `-ne`: Not equal
- `-lt`: Less than
- `-le`: Less than or equal
- `-gt`: Greater than
- `-ge`: Greater than or equal

Example: Check if a file has more than 100 lines:

```
if [ $(wc -l < file.txt) -gt 100 ]; then
    echo "Large file!"
fi
```

File and Directory Tests

Shell scripts often need to check if files exist:

```
if [ -f "myfile.txt" ]; then
    echo "File exists and is a regular file"
elif [ -d "myfile.txt" ]; then
    echo "It's a directory, not a file"
elif [ -e "myfile.txt" ]; then
    echo "Something exists with that name"
else
    echo "File does not exist"
fi
```

Common file tests:

- `-f file`: Regular file exists
- `-d file`: Directory exists
- `-e file`: File or directory exists
- `-r file`: File is readable
- `-w file`: File is writable
- `-x file`: File is executable

🟦 Loops: Repeating Actions

For loops process lists of items:

```
cat my_project/file_processor.sh
```

──────────────────── [finished] ────────────────────

```
#!/usr/bin/env bash
# Script that processes files in a directory

DIRECTORY=${1:-.}  # Use first argument or current
directory

echo "Processing files in: $DIRECTORY"
echo "===================="

for file in "$DIRECTORY"/*.txt; do
    if [ -f "$file" ]; then
        echo "File: $(basename "$file")"
        echo "  Lines: $(wc -l < "$file")"
        echo "  Words: $(wc -w < "$file")"
        echo "  Size: $(ls -lh "$file" | awk '{print
$5}')"
        echo ""
    fi
done

echo "Processing complete!"
```

This script demonstrates:

- **Default values**: `${1:-.}` uses first argument or current directory
- **For loop**: Processes each `.txt` file
- **File testing**: Checks if each item is actually a file
- **Command substitution**: Gets file statistics

```
chmod +x my_project/file_processor.sh
./my_project/file_processor.sh
```

Try this in a new terminal: Run the script to see it process files in the current directory.

Common patterns:

- Loop over arguments: `for arg in "$@"`
- Loop over files: `for file in *.txt`
- While loops for counters and reading input

Functions and error handling:

- Use `local` variables in functions
- `set -e` to exit on errors
- Always check important return values

For detailed examples, loops, functions, and debugging techniques, see the appendix slides.

Key principles:

- Always quote variables: `"$VARIABLE"`
- Use meaningful variable names
- Include error checking
- Comment your code

For more advanced scripting techniques, debugging, and best practices, see the appendix slides.

Practice suggestions:

- Start with simple automation tasks
- Use ShellCheck to validate your scripts
- Study existing scripts to learn patterns

Remember: Shell scripting is powerful for automation, but consider Python for complex logic.

For complete scripting examples, advanced techniques, and debugging, see the appendix slides.

Everyday Skills: Managing Your System

Now let's learn essential day-to-day skills for working effectively in Linux:

What we'll cover:

- Controlling processes and programs
- Managing disk space and storage
- Working with background tasks
- Multiplexing your terminal sessions

These are the skills that separate casual users from power users!

Understanding Processes: What's Running?

Linux runs multiple processes simultaneously. Let's see what's happening:

```
ps
```

[finished]

```
    PID TTY          TIME CMD
2406170 pts/5    00:00:03 zsh
2413666 pts/5    00:00:01 zsh
2415270 pts/5    00:00:00 presenterm
2415439 pts/5    00:00:00 bash
2415441 pts/5    00:00:00 bash
2415483 pts/5    00:00:00 ps
2415484 pts/5    00:00:00 ps
2415485 pts/5    00:00:00 head
```

`ps`: Shows running processes in your current terminal session

```
ps aux | head -10
```

[finished]

```
USER         PID %CPU %MEM    VSZ   RSS TTY      STAT
START   TIME COMMAND
root           1  0.0  0.0  25496 10652 ?        Ss
Aug26   0:45
/run/current-system/systemd/lib/systemd/systemd
root           2  0.0  0.0      0     0 ?        S
Aug26   0:00 [kthreadd]
root           3  0.0  0.0      0     0 ?        S
Aug26   0:00 [pool_workqueue_release]
root           4  0.0  0.0      0     0 ?        I<
Aug26   0:00 [kworker/R-kvfree_rcu_reclaim]
root           5  0.0  0.0      0     0 ?        I<
Aug26   0:00 [kworker/R-rcu_gp]
root           6  0.0  0.0      0     0 ?        I<
Aug26   0:00 [kworker/R-sync_wq]
root           7  0.0  0.0      0     0 ?        I<
```

Process control essentials:

- `top`: Interactive process monitor (`q` to quit)
- `jobs`: Show background processes
- `kill PID`: Terminate process
- `kill -9 PID`: Force terminate (last resort)
- `Ctrl+C`: Interrupt current process
- `&`: Run command in background

For detailed process management and signal handling, see the appendix slides.

🟦 Managing Disk Space: Know Your Usage

Disk space management is crucial for system health:

```
df -h
```

——————————————————— [finished] ———————————————————

```
Filesystem
Size  Used Avail Use% Mounted on
devtmpfs
788M     0  788M   0% /dev
tmpfs
7.7G   87M  7.7G   2% /dev/shm
tmpfs
3.9G  6.3M  3.9G   1% /run
/dev/dm-0
221G  160G   54G  75% /
efivarfs
154K   85K   65K  57% /sys/firmware/efi/efivars
tmpfs
1.0M     0  1.0M   0%
/run/credentials/systemd-journald.service
tmpfs
1.0M     0  1.0M   0%
/run/credentials/systemd-cryptsetup@swap.service
/dev/nvme0n1p1
133M   54M   79M  41% /boot
tmpfs
7.7G  1.3M  7.7G   1% /run/wrappers
tmpfs
1.6G   23M  1.6G   2% /run/user/1000
/dev/mapper/tomb..password.b12ad4103f29be8d89fb4f605e57f
5a93bbd9ab6cb21b9e36ffaaa12499ae4a6.loop0  3.8M  1.9M
1.4M  59% /home/chris/.local/share/password-store
tmpfs
1.0M     0  1.0M   0%
/run/credentials/getty@tty2.service
```

`df -h`: Shows **d**isk **f**ree space in **h**uman-readable format

What the columns mean:

- **Filesystem**: Storage device or partition

🟦 Finding What's Using Your Disk Space

When disk space is low, find the culprits:

```
du -h . | head -10
```

──────────────────── [finished] ────────────────────

```
180K    ./old/img
236K    ./old
1.8M    ./img
20K    ./my_project
4.0K    ./Documents
4.0K    ./logs
4.0K    ./config
0    ./practice
3.9M    .
```

`du -h`: Shows **d**isk **u**sage in **h**uman-readable format

```
du -sh *
```

──────────────────── [finished] ────────────────────

```
16K    am215_lec01_linux_appendix.md
64K    am215_lec01_linux.md
1.8M    am215_lec01_linux.pdf
4.0K    config
4.0K    Documents
1.8M    img
4.0K    logs
20K    my_project
236K    old
0    practice
4.0K    sample_data.txt
4.0K    temp_file.txt
```

This shows the total size of each item in the current directory.

Useful `du` patterns:

Finding Large Files

Sometimes individual files are the problem:

```
find . -type f -size +1M | head -5
```

[finished]

```
./am215_lec01_linux.pdf
```

Find files larger than 1 megabyte in the current directory.

Common size units:

- `c`: bytes
- `k`: kilobytes
- `M`: megabytes
- `G`: gigabytes

Useful file-finding patterns:

```
# Files larger than 100MB
find /home -type f -size +100M

# Files modified in the last 7 days
find . -type f -mtime -7

# Old log files (older than 30 days)
find /var/log -name "*.log" -mtime +30
```

Cleanup strategy: Look for old downloads, logs, and temporary files first.

Background tasks and multiplexing:

- `nohup command &`: Run task that survives logout
- `screen`: Multiple terminal sessions in one window
- `tmux`: Modern alternative to screen with better features

Job control:

- `fg %1`: Bring background job to foreground
- `bg %1`: Send suspended job to background

For detailed multiplexing tutorials and advanced usage, see the appendix slides.

🟦 System Monitoring: Keeping an Eye on Things

Monitor system health with built-in tools:

```
uptime
```

──────────────────── [finished] ────────────────────

```
 02:25:03  up 9 days 12:54,  2 users,  load average:
1.43, 1.68, 1.54
```

Shows how long the system has been running and current load.

```
free -h
```

──────────────────── [finished] ────────────────────

```
               total        used        free      shared
buff/cache   available
Mem:            15Gi       7.8Gi       2.3Gi       1.2Gi
5.2Gi       7.6Gi
Swap:           17Gi       1.9Gi        15Gi
```

Shows memory usage in human-readable format.

Key metrics to watch:

- **Load average**: Should be less than number of CPU cores
- **Memory usage**: Swap usage indicates memory pressure
- **Disk space**: Keep filesystems under 90% full

Quick health check script:

```
echo "=== System Health ==="
uptime
echo ""
df -h | grep -E "(Filesystem|/dev/)"
echo ""
free -h
```

File compression basics:

```
tar -czf backup.tar.gz directory/    # Create archive
tar -xzf backup.tar.gz               # Extract archive
```

Common formats: `.tar.gz` (general use), `.zip` (cross-platform)

For advanced compression and networking commands, see the appendix slides.

Putting It All Together: Daily Workflow

A typical power-user workflow combining these skills:

```
# Morning system check
uptime && df -h && free -h

# Start main work session
tmux new-session -s work

# In tmux: create windows for different tasks
# Ctrl+B, C (new window for editing)
# Ctrl+B, C (new window for testing)
# Ctrl+B, C (new window for monitoring)

# Start long-running background task
nohup ./data_processing.sh > processing.log 2>&1 &

# Monitor progress occasionally
tail -f processing.log

# When switching projects
# Ctrl+B, D (detach tmux)
tmux new-session -s project2

# End of day: check what's still running
jobs
ps aux | grep myusername
```

This workflow demonstrates: Process management, multiplexing, background tasks, and monitoring all working together.

Troubleshooting Common Issues

Solutions to everyday problems:

System running slowly:

```
top                       # Find CPU-hungry processes
free -h                  # Check memory usage
df -h                    # Check disk space
```

Can't find a file:

```
find / -name "filename" 2>/dev/null
locate filename       # If locate database exists
```

Process won't stop:

```
ps aux | grep process_name
kill -15 PID          # Try gentle termination
kill -9 PID           # Force kill if needed
```

Disk space full:

```
du -sh /* | sort -hr  # Find largest directories
find /tmp -type f -mtime +7 -delete  # Clean old temp files
```

Remember: Always understand what a command does before running it, especially with `rm` or `kill`!

🟦 Other Shells: Beyond Bash

While we've focused on bash, there are many other shells available:

```
echo $SHELL
```

——————————————————— [finished] ———————————————————

```
/run/current-system/sw/bin/zsh
```

Shows your current shell.

Popular shell alternatives:

- **zsh** (Z Shell): Enhanced bash with better completion and themes
- **fish** (Friendly Interactive Shell): User-friendly with syntax highlighting
- **dash**: Minimal POSIX shell, very fast
- **tcsh**: C-style shell syntax
- **ksh**: Korn shell, POSIX compliant

Why different shells?

- Personal preference and workflow
- Specific features (auto-completion, syntax highlighting)
- Performance requirements
- Compatibility needs

🟦 Choosing Your Shell

Factors to consider when choosing a shell:

Stick with Bash if:

- You're learning Linux fundamentals
- You need maximum compatibility
- You work with many different systems
- You write lots of shell scripts

Try Zsh if:

- You want bash compatibility with enhancements
- You like customization and themes
- You want better completion and correction
- You use macOS (it's the default)

Try Fish if:

- You prioritize user experience
- You want modern features out of the box
- You don't mind learning different syntax
- You do more interactive work than scripting

Remember: You can always change shells later with `chsh -s /path/to/shell`

CLI vs TUI vs GUI: Interface Evolution

Three main ways to interact with computers:

CLI (Command Line Interface):

- Text-based commands
- Maximum efficiency for experts
- Scriptable and automatable
- Universal across systems

TUI (Text User Interface):

- Text-based but with menus and forms
- Examples: `htop`, `nano`, `midnight commander`
- More discoverable than CLI
- Still works over SSH

GUI (Graphical User Interface):

- Windows, icons, mouse interaction
- Most intuitive for beginners
- Rich visual feedback
- Requires graphics system

When to Use Each Interface

CLI is best for:

- Repetitive tasks that can be automated
- Remote server administration
- Complex data processing pipelines
- When you know exactly what you want to do
- Scripting and automation

TUI is best for:

- Interactive system monitoring (`htop`, `iotop`)
- File management (`mc`, `ranger`)
- Text editing (`nano`, `vim`)
- When you need menus but want to stay in terminal

GUI is best for:

- Visual tasks (image editing, web browsing)
- Learning new software
- Complex layouts and multiple windows
- When you need to see visual feedback

The power user approach: Use all three as appropriate!

The Philosophy of Unix Interfaces

Why does Linux emphasize text interfaces?

Historical reasons:

- Computers were expensive, terminals were cheap
- Network bandwidth was limited
- Graphics were primitive or nonexistent

Technical advantages:

- **Composability**: Text output can be piped to other programs
- **Scriptability**: Automate any task you can do manually
- **Efficiency**: No graphics overhead
- **Universality**: Works the same everywhere

Modern relevance:

- **Cloud computing**: Most servers have no GUI
- **Automation**: DevOps and system administration
- **Efficiency**: Faster for many tasks once learned
- **Accessibility**: Works over slow connections

Modern Linux Desktop Environments

Linux isn't just about the command line!

Popular desktop environments:

- **GNOME**: Modern, clean interface (Ubuntu default)
- **KDE Plasma**: Highly customizable, Windows-like
- **XFCE**: Lightweight, traditional desktop
- **i3/Sway**: Tiling window managers for power users

The beauty of Linux: You can choose your interface level:

- Pure command line (servers)
- Minimal window manager + terminal
- Full desktop environment
- Mix and match as needed

Best of both worlds: Modern Linux desktops integrate CLI tools seamlessly with GUI applications.

🟦 Why Learn the Command Line in 2024?

"Isn't the command line obsolete?"

Absolutely not! Here's why:

Professional relevance:

- **Cloud computing**: Most servers are Linux without GUI
- **DevOps**: Automation requires scripting skills
- **Data science**: Many tools are command-line first
- **Development**: Git, package managers, build tools

Efficiency gains:

- **Batch operations**: Process hundreds of files at once
- **Remote work**: Manage systems over SSH
- **Automation**: Script repetitive tasks
- **Troubleshooting**: Direct access to system internals

Universal skills:

- **Cross-platform**: Similar commands on macOS, Linux, WSL
- **Timeless**: Core concepts haven't changed in decades
- **Transferable**: Skills apply to many tools and systems

What We've Learned Today

From history to hands-on skills:

Unix/Linux History:

- Bell Labs origins and the Unix philosophy
- GNU Project and free software movement
- Linux kernel and the complete GNU/Linux system

Terminal Fundamentals:

- User identity, file system navigation
- Environment variables and shell customization
- File operations and permissions

Text Processing:

- Pipes, redirection, and the Unix philosophy
- Regular expressions and pattern matching
- sed and awk for advanced text manipulation

Shell Scripting:

- Automation through scripts
- Variables, conditionals, and loops
- Best practices and debugging

Everyday Skills:

- Process management and system monitoring
- Background tasks and terminal multiplexing
- Troubleshooting common issues

Next Steps: Continuing Your Linux Journey

Immediate practice:

1. **Set up your environment**: Customize your `~/.bashrc`
2. **Daily usage**: Try to use terminal for file operations
3. **Write scripts**: Automate a repetitive task
4. **Explore tools**: Try `tmux`, different editors, alternative shells

Intermediate goals:

- **System administration**: Learn about services, logs, networking
- **Package management**: Master your distribution's package manager
- **Version control**: Integrate Git into your workflow
- **Remote access**: Learn SSH and remote system management

Advanced topics:

- **System programming**: Understand how Linux works internally
- **Container technology**: Docker, Kubernetes
- **Infrastructure as code**: Ansible, Terraform
- **Performance tuning**: Optimize systems for specific workloads

Remember: Linux mastery is a journey, not a destination. Every expert was once a beginner!

Essential references:

- **Man pages**: `man command` - always your first stop
- **Appendix slides**: Advanced topics, detailed examples, and comprehensive tutorials
- **The Linux Documentation Project**: **tldp.org** (*https://tldp.org*)

Practice environments:

- **Virtual machines**: Safe place to experiment
- **Cloud instances**: AWS, Google Cloud, DigitalOcean
- **WSL**: Windows Subsystem for Linux

Final Thoughts: The Unix Philosophy Lives On

"Do one thing and do it well"

The Unix philosophy we learned about today continues to influence modern computing:

In software development:

- Microservices architecture
- API design principles
- Container technology

In data processing:

- Stream processing systems
- ETL pipelines
- Big data tools

In system design:

- Cloud-native applications
- Infrastructure as code
- DevOps practices

The command line isn't just about Linux - it's about understanding how to build composable, scriptable, efficient systems.

Your journey starts now. Every command you run, every script you write, every problem you solve makes you more capable of working with the systems that power our digital world.

Welcome to the command line. Welcome to Linux. Welcome to a more powerful way of computing.

🟦 Big Picture Takeaways

What makes Linux/Unix special:

- **Composability**: Small tools that work together through pipes and redirection
- **Scriptability**: Everything you can do manually, you can automate
- **Transparency**: Text-based interfaces reveal how systems actually work
- **Durability**: Core concepts from the 1970s still power modern computing

Skills that transfer everywhere:

- **Problem decomposition**: Breaking complex tasks into simple, chainable steps
- **Automation mindset**: Recognizing repetitive work and scripting solutions
- **System thinking**: Understanding how components interact and communicate
- **Debugging approach**: Using logs, processes, and tools to diagnose issues

The real power isn't in memorizing commands - it's in understanding the philosophy of building reliable, composable, automatable systems.

This foundation will serve you whether you're:

- Managing cloud infrastructure
- Processing research data
- Building software systems
- Analyzing large datasets

The command line is not just a tool - it's a way of thinking about computing.