

Course Overview and The Shell

CMSC398W: Practical Tools For Efficient Development

Mohammad Durrani

February 4, 2026

Welcome To Class!

Icebreaker

Please share in the Zoom chat:

- ① Your year (Freshman, Sophomore, Junior, Senior)
- ② Either:
 - One thing you did over break, OR
 - What you hope to learn from this class

About The Class

Course Information

CMSC398W: Practical Tools For Efficient Development

- Overview of tools used in development: command line, Git, debuggers, build systems, etc.
- Emphasis on breadth over depth
- Hands-on learning through projects

Course Goals

- Improve your computing ecosystem literacy
- Improve your efficiency
- Reduce cognitive load while developing

About Me - Mohammad

Mohammad Durrani

- Senior
- CS + Math, minor in Robotics
- Previous Experience:
 - Teaching this class since Spring 2025
 - SWE Intern at Google in SF
 - SWE Intern at TRX Systems in Greenbelt

Hobbies:

- Rock climbing
- Basketball
- Robots

What To Expect

Course Structure

Read the syllabus for all course-related information!

- Introduction of topic
- Motivation / real-world example
- Technical details
- Projects focusing on application
- Best way to learn: **use them**

Course Logistics

Communication & Submission

- All course communication: **Piazza**
- All assignments submitted: **Gradescope**
- **10% per-day late penalty** (except the last project)

Grade Breakdown

Percentage	Title	Description
80%	Projects (20% per)	4 Major Projects
15%	Application Days (5% per)	Completion of Application Days
5%	Participation	Participation in class

Application Days

Based on Feedback

Based on feedback from the previous iteration, we'll have days focused on using the tools we learn in class on toy problems.

Application Days

Based on Feedback

Based on feedback from the previous iteration, we'll have days focused on using the tools we learn in class on toy problems.

Three Application Days

- ① Shell
- ② Git
- ③ Networking

(Subject to change)

Grade Distribution



What is the Shell?

Definition (Shell)

A text-based interface to the operating system.

What is the Shell?

Definition (Shell)

A text-based interface to the operating system.

Terminal vs Shell

Component	Role	Examples
Terminal	The application that runs the shell	iTerm2, Windows Terminal
Shell	Interprets and runs commands	bash, zsh, fish

Why Do We Need It?

Power & Efficiency

- **Speed:** Automate repetitive tasks

Why Do We Need It?

Power & Efficiency

- **Speed:** Automate repetitive tasks
- **Control:** Do things GUIs simply can't

Why Bash?

We focus on **Bash** because it's common and the skills are largely transferable. On macOS the default interactive shell is often zsh, but the core concepts and scripting translate.

Why Do We Need It?

Power & Efficiency

- **Speed:** Automate repetitive tasks
- **Control:** Do things GUIs simply can't
- **Remote Machines:** The standard way to manage servers

Why Bash?

We focus on **Bash** because it's common and the skills are largely transferable. On macOS the default interactive shell is often zsh, but the core concepts and scripting translate.

The Prompt

Anatomy of the Prompt

username@host:directory\$

Component	Description
username	Currently logged in user
@	Separator (convention)
host	Machine name
directory	Current working directory
\$	User-level shell indicator
#	Root-level shell indicator

Basic Commands

How It Works

Type command shell splits on whitespace runs program with arguments

Prompts vary and are customizable (your PS1 controls this).

Basic Commands

How It Works

Type command shell splits on whitespace runs program with arguments

Prompts vary and are customizable (your PS1 controls this).

Examples

```
date          # Show date/time  
echo hello    # Print text  
echo "Hello World" # Handle spaces  
ls -l ~       # Command with flags
```

File Paths

Types

Absolute: /home/user/docs (from root)

Relative: ../other/file.txt (from current location)

File Paths

Types

Absolute: /home/user/docs (from root)

Relative: ../other/file.txt (from current location)

Special Symbols

Symbol	Meaning
.	Current directory
..	Parent directory
~	Home directory

Essential Navigation Commands

Why?

How do you know where you are in a system with millions of files?

Essential Navigation Commands

Why?

How do you know where you are in a system with millions of files?

Commands

Command	Description
pwd	Print working directory
cd <dir>	Change directory
ls [dir]	List directory contents
ls -l	Long format listing
ls -a	Show hidden files
ls -lh	Human-readable sizes

Navigation Examples

Our Filesystem Structure

```
/ (root)
bin/
home/
    user/
        docs/
        photos/
usr/
```

Navigation Examples

Our Filesystem Structure

```
/ (root)
bin/
home/
    user/
        docs/
        photos/
usr/
```

Using Navigation Commands

```
$ pwd
/home/user
```

```
$ ls -F          # -
F adds / to directories
docs/  photos/
```

```
$ cd docs
$ pwd
/home/user/docs
```

```
$ cd ..
$ pwd
/home/user
```

Think-Pair-Share: Navigation Puzzle

The Path

You start in `/home/user/docs`. You run the following command:

```
cd ../../photos/../../docs/.../
```

The Question

Where are you now? Work with a partner to trace the path step-by-step.

Think-Pair-Share: Navigation Puzzle

The Path

You start in `/home/user/docs`. You run the following command:

```
cd ../../photos/../../docs/.../
```

The Question

Where are you now? Work with a partner to trace the path step-by-step.

Answer

`/home`

- ① `..` → `/home/user`
- ② `./photos` → `/home/user/photos`

Think-Pair-Share: Navigation Puzzle

The Path

You start in `/home/user/docs`. You run the following command:

```
cd ../../photos/../../docs/.../
```

The Question

Where are you now? Work with a partner to trace the path step-by-step.

Answer

`/home`

- ① `..` → `/home/user`
- ② `./photos` → `/home/user/photos`
- ③ `..` → `/home/user`

Think-Pair-Share: Navigation Puzzle

The Path

You start in `/home/user/docs`. You run the following command:

```
cd ../../photos/../../docs/../../
```

The Question

Where are you now? Work with a partner to trace the path step-by-step.

Answer

`/home`

- ① `..` → `/home/user`
- ② `./photos` → `/home/user/photos`
- ③ `..` → `/home/user`
- ④ `../../docs` → `/home/user/docs`

Think-Pair-Share: Navigation Puzzle

The Path

You start in `/home/user/docs`. You run the following command:

```
cd ../../photos/../../docs/.../
```

The Question

Where are you now? Work with a partner to trace the path step-by-step.

Answer

`/home`

- ① `..` → `/home/user`
- ② `./photos` → `/home/user/photos`
- ③ `..` → `/home/user`
- ④ `../../docs` → `/home/user/docs`
- ⑤ `.../.../` → `/home`

File and Directory Manipulation

Common Operations

Command	Description
<code>mkdir <dir></code>	Make directory
<code>touch <file></code>	Create empty file / update timestamp
<code>cp <src> <dst></code>	Copy files/directories
<code>mv <src> <dst></code>	Move/rename files
<code>rm <file></code>	Remove files

File and Directory Manipulation

Common Operations

Command	Description
<code>mkdir <dir></code>	Make directory
<code>touch <file></code>	Create empty file / update timestamp
<code>cp <src> <dst></code>	Copy files/directories
<code>mv <src> <dst></code>	Move/rename files
<code>rm <file></code>	Remove files

What do these flags actually do?

- `-r`: Recursive (directories)
- `-i`: Interactive (ask before delete)

File and Directory Manipulation

Common Operations

Command	Description
<code>mkdir <dir></code>	Make directory
<code>touch <file></code>	Create empty file / update timestamp
<code>cp <src> <dst></code>	Copy files/directories
<code>mv <src> <dst></code>	Move/rename files
<code>rm <file></code>	Remove files

What do these flags actually do?

- `-r`: Recursive (directories)
- `-i`: Interactive (ask before delete)
- `-f`: Force (no prompt)

File and Directory Manipulation

Common Operations

Command	Description
<code>mkdir <dir></code>	Make directory
<code>touch <file></code>	Create empty file / update timestamp
<code>cp <src> <dst></code>	Copy files/directories
<code>mv <src> <dst></code>	Move/rename files
<code>rm <file></code>	Remove files

What do these flags actually do?

- `-r`: Recursive (directories)
- `-i`: Interactive (ask before delete)
- `-f`: Force (no prompt)
- `-v`: Verbose (show progress)

File Manipulation Examples

Examples

```
# Create nested directories  
mkdir -p project/src/utils
```

```
# Copy a folder and all its contents  
cp -r folder1 folder1_backup
```

```
# Rename a file  
mv old_name.txt new_name.txt
```

```
# Remove a folder and its contents (be careful!)  
rm -rf temporary_work
```

Time to Explore: File Operations

Challenge

Perform the following tasks using only the shell:

- ① Create a directory named STIC, and inside it, a directory named lab1.
- ② Create three empty files in lab1 named test1.py, test2.py, and notes.txt.
- ③ Copy notes.txt to a new file called README.md.
- ④ Move all .py files into a new subdirectory called src.
- ⑤ Try to remove the STIC directory using rmdir. Why does it fail?

Solutions: File Operations

Steps

```
# 1. Create nested  
mkdir -p STIC/lab1
```

```
# 2. Create files  
touch STIC/lab1/test1.py STIC/lab1/test2.py STIC/lab1/notes.txt
```

```
# 3. Copy  
cp STIC/lab1/notes.txt STIC/lab1/README.md
```

```
# 4. Move  
mkdir STIC/lab1/src  
mv STIC/lab1/*.py STIC/lab1/src/
```

```
# 5. Why fail?  
rmdir STIC # Fails: "Directory not empty"  
rm -r STIC # Use recursive remove instead!
```

Viewing File Contents

Why not just use VS Code?

What if the file is on a remote server? What if it's 10GB?

Viewing File Contents

Why not just use VS Code?

What if the file is on a remote server? What if it's 10GB?

Viewing Commands

Command	Description
cat <file>	Display entire file
less <file>	Page through file (q to quit)
head -n N <file>	Show first N lines
tail -n N <file>	Show last N lines
tail -f <file>	Follow file updates (logs)

Viewing Examples

File Viewing

```
# Display entire file  
cat file.txt
```

```
# Page through file (Search with /, quit with q)  
less file.txt
```

```
# First 20 lines  
head -n 20 file.txt
```

```
# Last 15 lines  
tail -n 15 file.txt
```

```
# Follow log file as it grows  
tail -f /var/log/system.log
```

Think-Pair-Share: Which tool when?

Match the tool to the task

- ① You want to see the last 5 errors in a log file.
- ② You want to read a 1GB text file without crashing your computer.
- ③ You want to see the first line of every .txt file in a directory.
- ④ You want to watch a log file update in real-time as you run a server.

Think-Pair-Share: Which tool when?

Match the tool to the task

- ① You want to see the last 5 errors in a log file.
- ② You want to read a 1GB text file without crashing your computer.
- ③ You want to see the first line of every .txt file in a directory.
- ④ You want to watch a log file update in real-time as you run a server.

Answers

- ① tail -n 5
- ② less (it doesn't load the whole file at once!)

Think-Pair-Share: Which tool when?

Match the tool to the task

- ① You want to see the last 5 errors in a log file.
- ② You want to read a 1GB text file without crashing your computer.
- ③ You want to see the first line of every .txt file in a directory.
- ④ You want to watch a log file update in real-time as you run a server.

Answers

- ① `tail -n 5`
- ② `less` (it doesn't load the whole file at once!)
- ③ `head -n 1 *.txt`

Think-Pair-Share: Which tool when?

Match the tool to the task

- ① You want to see the last 5 errors in a log file.
- ② You want to read a 1GB text file without crashing your computer.
- ③ You want to see the first line of every .txt file in a directory.
- ④ You want to watch a log file update in real-time as you run a server.

Answers

- ① `tail -n 5`
- ② `less` (it doesn't load the whole file at once!)
- ③ `head -n 1 *.txt`
- ④ `tail -f`

PATH: Finding Programs

How does the shell know where 'ls' is?

The PATH environment variable is a list of directories the shell searches through every time you type a command.

Key Points

- Colon-separated list: /usr/local/bin:/usr/bin:/bin
- **First match wins:** The shell searches from left to right.
- If it's not in PATH, use an explicit path (e.g., /usr/local/bin/my_prog or ./my_prog).

PATH Variable - Example

Exploring PATH

```
# Display your PATH  
$ echo $PATH  
/usr/local/bin:/usr/bin:/bin:/usr/sbin:/sbin
```

```
# Find where a command actually lives (portable)  
$ command -v ls  
/bin/ls
```

```
# Run directly without relying on PATH  
$ /bin/date  
# sample output: Fri Jan 30 14:00:00 EST 2026
```

Modifying PATH

Adding to PATH

```
export PATH="$HOME/my_tools:$PATH"
```

This adds `my_tools` to the beginning of your PATH, giving it priority.

Modifying PATH

Adding to PATH

```
export PATH="$HOME/my_tools:$PATH"
```

This adds `my_tools` to the beginning of your PATH, giving it priority.

Rhetorical Question

Why might it be dangerous to add `.` (current directory) to the **start** of your PATH?

Environment Variables

Definition (Definition)

Variables that define the environment in which your programs run.

Common Variables

Var	Meaning
PATH	Command search
HOME	Home directory
USER	Username
PWD	Current dir

Usage Examples

```
# Display  
echo $USER
```

```
# Set (local to shell)  
MY_VAR="value"
```

```
# Export (available to child procs)  
export MY_VAR="value"
```

```
# View all  
env | head
```

Time to Explore: Environment

Challenge

- ① Type `env` and look at the output. Can you find `SHELL` and `PWD`?
- ② Create a variable: `MY_NAME`"Your Name". Try `=echo $MY_NAME`.
- ③ Open a **new** terminal window. Is `$MY_NAME` still there?
- ④ Try `export PS1="Ready> "=` (or something fun). What happened to your prompt?

Solutions: Environment

Key Takeaways

- **Ephemeral:** Variables die when the terminal is closed.
- **Local vs Export:** Without export, child programs (like scripts you run) can't see the variable.

Solutions: Environment

Key Takeaways

- **Ephemeral:** Variables die when the terminal is closed.
- **Local vs Export:** Without export, child programs (like scripts you run) can't see the variable.
- **Customization:** PS1 is the variable that controls what your prompt looks like!

Searching and Finding

Why search from the CLI?

Imagine searching through 100,000 lines of logs or finding a specific file in a project with 50 subdirectories.

Searching and Finding

Why search from the CLI?

Imagine searching through 100,000 lines of logs or finding a specific file in a project with 50 subdirectories.

Search Commands

Command	Description	Useful Flags
find	Find files	-name, -type, -size
grep	Search text	-r (recursive), -i (case), -n (line#)
which	Find exe	-

find: Locate Files

Examples with Output

```
$ find . -name "*.txt"
```

```
./notes.txt
```

```
./data/results.txt
```

```
# Find directories only
```

```
$ find . -type d -name "test*"
```

```
./project/tests
```

```
# Find files larger than 100MB
```

```
$ find /var/log -size +100M
```

grep: Search File Contents

Examples

```
$ grep "error" app.log  
Connection error at 10:32
```

```
# Show line numbers (-n)  
$ grep -n "error" app.log  
15:Connection error at 10:32
```

```
# Recursive search (-r) in a directory  
$ grep -r "TODO" src/  
src/main.py: # TODO: optimize this
```

```
# Case-insensitive (-i)  
$ grep -i "ERROR" app.log
```

Think-Pair-Share: The Needle in the Haystack

The Challenge

You are working on a massive project. You need to find a "TODO" comment related to "authentication" in any file.

Task: How would you find this line using only grep? Discuss with your neighbor the flags you'd need.

Think-Pair-Share: The Needle in the Haystack

The Challenge

You are working on a massive project. You need to find a "TODO" comment related to "authentication" in any file.

Task: How would you find this line using only grep? Discuss with your neighbor the flags you'd need.

Possible Solution

```
$ grep -rn "TODO.*authentication" .
```

- **-r**: Search all files in all subdirectories.
- **-n**: Tell me exactly which line the comment is on!

Wildcards and Globbing

Glob Patterns

Pattern	Matches
*	Any string (including empty)
?	Any single character
[abc]	Any character in brackets
[a-z]	Any character in range

Globbing Examples

Basic Wildcards

```
# All .txt files
```

```
ls *.txt
```

```
# Files starting with 'test'
```

```
ls test*
```

```
# Single character: matches file1.txt but not file10.txt
```

```
ls file?.txt
```

Globbing Examples - Advanced

Character Sets and Negation

```
# Match specific numbers
```

```
ls file[123].txt
```

```
# Match any lowercase letter
```

```
ls [a-z]*.txt
```

```
# Negation: anything NOT starting with a-z
```

```
ls [!a-z]*
```

Time to Explore: Globbing

Files in Directory

img1.png, img10.png, img2.png, image.png, backup_img1.png

Predict

Which files match these patterns?

- ① img?.png
- ② img*.png
- ③ img[1-9].png
- ④ *img1.png

Time to Explore: Globbing

Files in Directory

img1.png, img10.png, img2.png, image.png, backup_img1.png

Predict

Which files match these patterns?

- ① img?.png
- ② img*.png
- ③ img[1-9].png
- ④ *img1.png

Answers

- ① img1.png, img2.png
- ② img1.png, img10.png, img2.png

Time to Explore: Globbing

Files in Directory

img1.png, img10.png, img2.png, image.png, backup_img1.png

Predict

Which files match these patterns?

- ① img?.png
- ② img*.png
- ③ img[1-9].png
- ④ *img1.png

Answers

- ① img1.png, img2.png
- ② img1.png, img10.png, img2.png
- ③ img1.png, img2.png

Time to Explore: Globbing

Files in Directory

img1.png, img10.png, img2.png, image.png, backup_img1.png

Predict

Which files match these patterns?

- ① img?.png
- ② img*.png
- ③ img[1-9].png
- ④ *img1.png

Answers

- ① img1.png, img2.png
- ② img1.png, img10.png, img2.png
- ③ img1.png, img2.png
- ④ img1.png, backup_img1.png

Brace Expansion

Definition (Creating Multiple Arguments)

Brace expansion generates multiple strings from a pattern. Excellent for bulk creation.

Examples

```
# Create multiple files at once
touch file{1,2,3}.txt
# Creates: file1.txt, file2.txt, file3.txt

# Ranges
echo {1..10}
echo {a..z}

# Nested expansion
mkdir -p project/{src,test,docs}
```

Command History

History Features

Key/Command	Action
Up/Down arrows	Navigate history
history	Show command history
!!	Repeat last command
!grep	Execute last command starting with grep
Ctrl+R	Reverse search history (The MVP!)

History Examples

Using History

```
# View last 20 commands  
history 20
```

```
# Forgot sudo?  
$ ls /var/root  
ls: /var/root: Permission denied  
$ sudo !!
```

```
# Search history  
(Press Ctrl+R, then type 'ssh')  
(reverse-i-search) 'ssh': ssh user@umd.edu
```

Tab Completion

Productivity Booster

Tab completion saves time and prevents typos. If you aren't mashing the Tab key, you're working too hard.

How it works

- Press Tab once to complete a unique match.

Tab Completion

Productivity Booster

Tab completion saves time and prevents typos. If you aren't mashing the Tab key, you're working too hard.

How it works

- Press Tab once to complete a unique match.
- Press Tab twice to see all possibilities if it's ambiguous.

Tab Completion

Productivity Booster

Tab completion saves time and prevents typos. If you aren't mashing the Tab key, you're working too hard.

How it works

- Press Tab once to complete a unique match.
- Press Tab twice to see all possibilities if it's ambiguous.
- Example: cd /u[TAB]l[TAB]b[TAB] → cd /usr/local/bin/

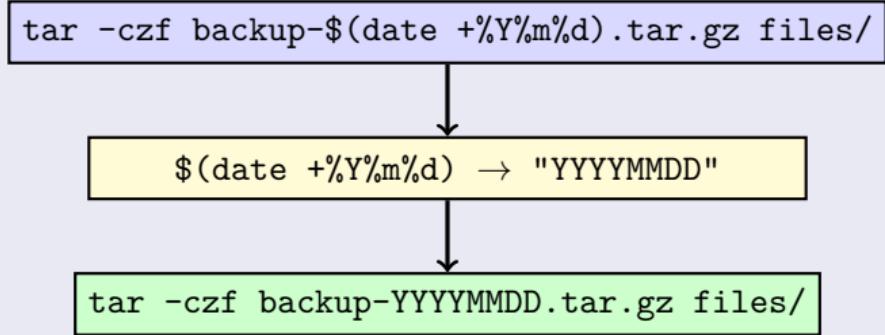
Command Substitution

Definition (Using Command Output)

Command substitution allows you to use the output of a command as an argument to another command.

Command Substitution Execution Order

How It Works



Inner command executes output replaces substitution full command runs

Command Substitution Examples

Substitution Syntax

```
# Using $(command) syntax (preferred)
echo "Today is $(date)"
files=$(ls -1)
echo "Found $(wc -l < file.txt) lines"

# Using backticks (older syntax)
echo "Today is `date`"

# Nested substitution
echo "User $(whoami) in $(pwd)"
```

Think-Pair-Share: Creative Substitution

Scenario

You want to create a directory named after the current year and month, and then move all your .log files into it.

Challenge

How can you do this in one or two lines using command substitution? (Hint: check `man date`)

Think-Pair-Share: Creative Substitution

Scenario

You want to create a directory named after the current year and month, and then move all your .log files into it.

Challenge

How can you do this in one or two lines using command substitution? (Hint: check `man date`)

Answer

```
mkdir "$(date +%Y-%m)"  
mv *.log "$(date +%Y-%m)"
```

Redirection Basics

Problem

Output goes to screen and disappears

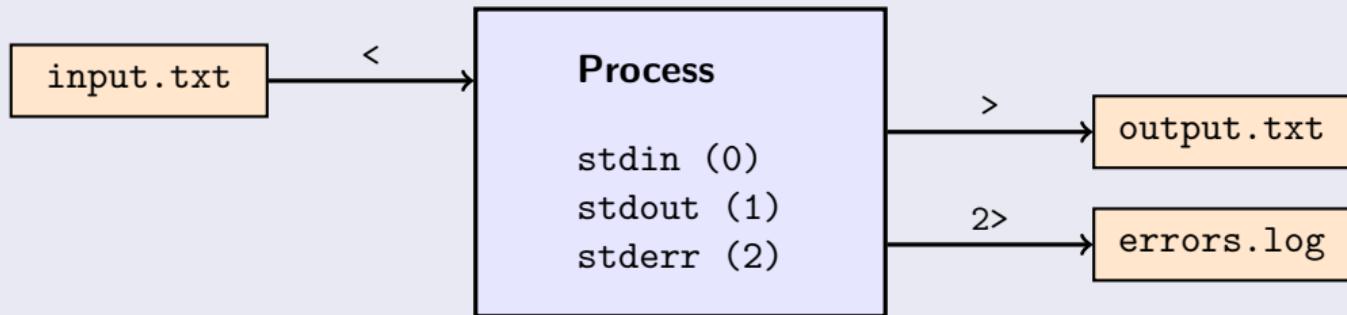
Solution

Operator	Action
>	Overwrite file
»	Append to file
2>	Redirect errors
&>	Redirect everything

Note: command &> file works in bash/zsh; the portable form is command > file 2>&1.

File Descriptors & Redirection

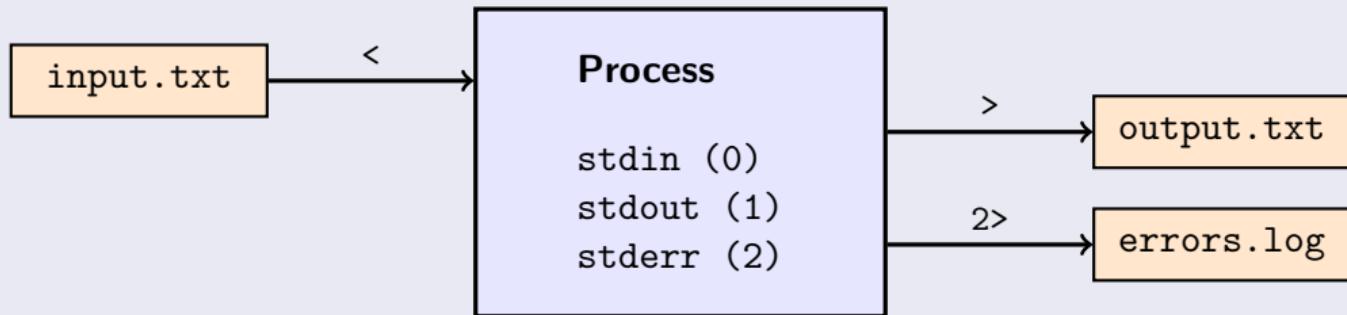
Visual Model



- command > file stdout to file

File Descriptors & Redirection

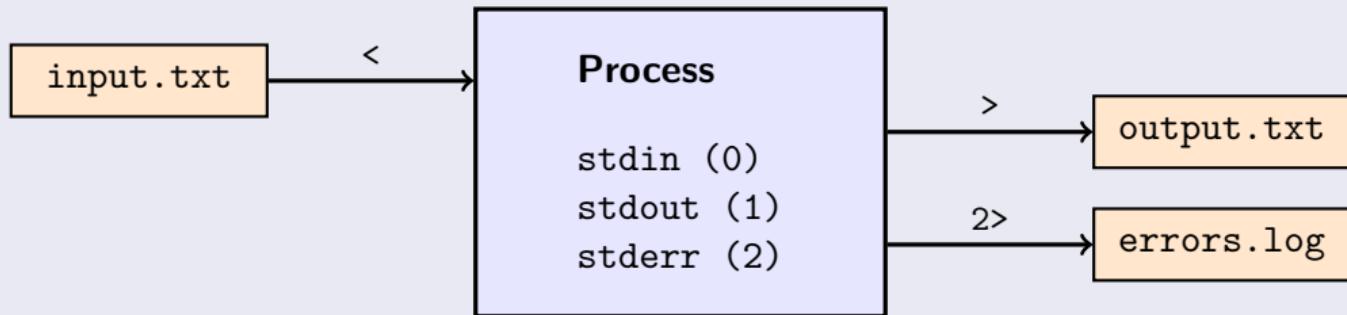
Visual Model



- command > file stdout to file
- command 2>&1 stderr to stdout

File Descriptors & Redirection

Visual Model



- command > file stdout to file
- command 2>&1 stderr to stdout
- command &> file both to file

Redirection Examples

Using Redirection

```
# Redirect output to file  
echo "Hello" > output.txt  
ls -l >> listing.txt
```

```
# Redirect input  
sort < unsorted.txt
```

```
# Redirect errors  
command 2> errors.log
```

```
# Combine stdout and stderr  
command > all.log 2>&1  
command &> all.log # Shorter syntax
```

Capturing Errors vs Output

Debugging Example

```
# Script that produces both stdout and stderr
$ python buggy_script.py
Processing file1.txt
ERROR: file2.txt not found
Done

# Capture separately
$ python buggy_script.py > output.txt 2> errors.txt

$ cat output.txt
Processing file1.txt
Done

$ cat errors.txt
ERROR: file2.txt not found
```

Think-Pair-Share: Redirection Pitfalls

Sequence

```
echo "Hello" > greeting.txt  
echo "World" >> greeting.txt  
echo "Goodbye" > greeting.txt  
cat greeting.txt
```

Discussion

- ① What is the final content of greeting.txt?
- ② What happens if you run `cat greeting.txt > greeting.txt`? **WARNING: Don't do this yet!**
- ③ How can you save both errors and output to the same file?

Think-Pair-Share: Redirection Pitfalls

Sequence

```
echo "Hello" > greeting.txt  
echo "World" >> greeting.txt  
echo "Goodbye" > greeting.txt  
cat greeting.txt
```

Discussion

- ➊ What is the final content of greeting.txt?
- ➋ What happens if you run cat greeting.txt > greeting.txt? **WARNING: Don't do this yet!**
- ➌ How can you save both errors and output to the same file?

Answer

- ➊ **Goodbye** (The final > nuked previous content).
- ➋ The file becomes **empty**! (The shell truncates the file for writing before cat can read it).

Think-Pair-Share: Redirection Pitfalls

Sequence

```
echo "Hello" > greeting.txt  
echo "World" >> greeting.txt  
echo "Goodbye" > greeting.txt  
cat greeting.txt
```

Discussion

- ➊ What is the final content of greeting.txt?
- ➋ What happens if you run cat greeting.txt > greeting.txt? **WARNING: Don't do this yet!**
- ➌ How can you save both errors and output to the same file?

Answer

- ➊ **Goodbye** (The final > nuked previous content).
- ➋ The file becomes **empty**! (The shell truncates the file for writing before cat can read it).
- ➌ command &> file

Pipes

Problem

Count .txt files:

- ls shows files
- grep filters
- wc -l counts

How to connect?

Solution

Use pipes (|)

```
ls -l | grep txt | wc -l
```

Pipes

Problem

Count .txt files:

- ls shows files
- grep filters
- wc -l counts

How to connect?

Solution

Use pipes (|)

```
ls -l | grep txt | wc -l
```

Philosophy

"Do one thing well" combine simple tools for complex tasks

How Pipes Work (Visual)

Data Flow



Each command runs simultaneously! Data flows left-to-right.
stdout of left becomes stdin of right.

Pipe Examples

Basic Piping

```
# Count entries in directory listing  
ls -1 | wc -l
```

```
# Search and sort  
cat file.txt | grep "pattern" | sort
```

```
# Find largest files  
du -h | sort -rh | head -10
```

```
# Process logs  
cat access.log | grep "404" | wc -l
```

Pipe Examples

Basic Piping

```
# Count entries in directory listing  
ls -1 | wc -l
```

```
# Search and sort  
cat file.txt | grep "pattern" | sort
```

```
# Find largest files  
du -h | sort -rh | head -10
```

```
# Process logs  
cat access.log | grep "404" | wc -l
```

Career Note

Data pipelines are used everywhere: ETL jobs, log analysis, build systems.

Building Pipes Step-by-Step

Progressive Example

```
# 1. See all lines
```

```
$ cat app.log
```

```
ERROR: connection failed
```

```
INFO: started successfully
```

```
ERROR: timeout
```

```
# 2. Filter errors only
```

```
$ cat app.log | grep ERROR
```

```
ERROR: connection failed
```

```
ERROR: timeout
```

```
# 3. Count errors
```

```
$ cat app.log | grep ERROR | wc -l
```

```
2
```

Think-Pair-Share: Debug the Pipeline

The Goal

Find all lines with "Error" in app.log and count them.

Broken Commands

Identify the flaw in each command:

- ① grep "Error" app.log > wc -l
- ② cat app.log | grep Error | wc

Think-Pair-Share: Debug the Pipeline

The Goal

Find all lines with "Error" in app.log and count them.

Broken Commands

Identify the flaw in each command:

- ① grep "Error" app.log > wc -l
- ② cat app.log | grep Error | wc

Answers

- ① **Redirection Error:** > sends output to a **file** named **wc**. Use | to send to a **program**.
- ② **Ambiguous Output:** **wc** shows lines, words, AND characters. Use **wc -l** for just lines.

xargs: Bridging the Gap

Definition (What is it?)

xargs builds and executes command lines from standard input. It converts lines of text into **arguments** for another command.

xargs: Bridging the Gap

Definition (What is it?)

xargs builds and executes command lines from standard input. It converts lines of text into **arguments** for another command.

Why do we need it?

Some commands (like rm, mkdir, mv) don't read from standard input. They only accept arguments. xargs bridges this gap.

xargs: Usage and Options

Examples

```
# Delete all .tmp files (safe with spaces)
find . -name "*.tmp" -print0 | xargs -0 rm
```

```
# Run 1 command per input line (NUL-delimited)
printf '%s\0' dir1 dir2 dir3 | xargs -0 -n 1 mkdir
```

```
# Custom placement of arguments (safe with spaces)
find . -maxdepth 1 -name "*.txt" -print0 | xargs -0 -I {} mv "{}" backup/
```

xargs: Usage and Options

Examples

```
# Delete all .tmp files (safe with spaces)
find . -name "*.tmp" -print0 | xargs -0 rm
```

```
# Run 1 command per input line (NUL-delimited)
printf '%s\0' dir1 dir2 dir3 | xargs -0 -n 1 mkdir
```

```
# Custom placement of arguments (safe with spaces)
find . -maxdepth 1 -name "*.txt" -print0 | xargs -0 -I {} mv "{}" backup/
```

Common Flags

Option	Description
-n N	Use at most N args per command
-I {}	Replace {} with the input string
-0	Read NUL-delimited input (safe)
-P N	Run N processes in parallel (Fast!)

System Health with top

Monitoring System Resources

top displays real-time system statistics: CPU, memory, running processes.

```
# macOS: Run top once and exit immediately  
$ top -l 1 -n 0
```

Flags: -l 1 runs top once (1 iteration). -n 0 displays 0 processes (header stats only).

System Health with top

Monitoring System Resources

top displays real-time system statistics: CPU, memory, running processes.

```
# macOS: Run top once and exit immediately  
$ top -l 1 -n 0
```

Flags: -l 1 runs top once (1 iteration). -n 0 displays 0 processes (header stats only).

Common Use Case

Capture CPU and memory usage in scripts for logging or alerting.

Section 3: Data Wrangling

The Goal

"Most of your time as a developer is spent moving data from one format to another."

Key Objectives

- Transform "messy" logs into structured reports.
- Clean and filter large datasets without opening heavy editors.
- Build automated pipelines for repetitive processing.

Data Wrangling Overview

Cleaning, transforming, and analyzing data using command-line tools.

Tasks:

- Extract fields
- Count frequencies
- Filter & Transform
- Reformat output

- **Filtering:** grep, head, tail
- **Sorting:** sort, uniq
- **Editing:** sed, awk, tr
- **Stats:** wc

Sorting and Uniqueness (sort, uniq)

Why?

uniq only works on **adjacent** lines. That's why we almost always sort before we uniq.

Sorting and Uniqueness (sort, uniq)

Why?

uniq only works on **adjacent** lines. That's why we almost always sort before we uniq.

Examples with Output

```
# Sample file: fruits.txt (apple, banana, apple, cherry, banana)

# Count unique occurrences
$ sort fruits.txt | uniq -c
    2 apple
    2 banana
    1 cherry

# Sort by frequency (descending)
$ sort fruits.txt | uniq -c | sort -rn
    2 banana
    2 apple
    1 cherry
```

Data Wrangling Example

Real-World Pipeline

```
# Analyze web server logs
# Find most common ERROR messages
cat access.log \
| grep "ERROR" \
| awk '{print $5}' \
| sort \
| uniq -c \
| sort -rn \
| head -10
```

Analysis

- ➊ **Read:** Opens the log file.
- ➋ **Filter:** Keeps only ERROR lines.
- ➌ **Extract:** Pulls the 5th column (the error type).
- ➍ **Sort:** Groups identical messages.
- ➎ **Count:** Counts unique occurrences.

Regular Expressions (Regex)

Why Regex?

Regex patterns allow for powerful, flexible text matching beyond simple wildcards.

Regex Flavors: A Shell Caveat

Comparison

Feature	BRE (Basic)	ERE (Extended)	PCRE (Perl/JS/Py)
Tools	grep, sed	grep -E, awk	grep -P, Python
+ , ?	Need \	Raw: + , ?	Raw: + , ?
() , { }	Need \	Raw: (), { }	Raw: (), { }

- **Note:** macOS BSD grep usually lacks -P. Prefer grep -E or install GNU grep (often ggrep).

Example: Matching "1 or more"

- **BRE:** grep 'a\+'
- **ERE:** grep -E 'a+'
- **PCRE:** grep -P 'a+'

Pattern Cheat Sheet

Type	Pattern	Matches
Anchors	^ / \$	Start / End of line
Quantify	* / +	Zero+ / One+ repetitions
Groups	[] / ()	Sets / Capture groups
Wildcard	.	Any single character

Pattern Cheat Sheet

Type	Pattern	Matches
Anchors	^ / \$	Start / End of line
Quantify	* / +	Zero+ / One+ repetitions
Groups	[] / ()	Sets / Capture groups
Wildcard	.	Any single character

Basic Examples

- ^apple matches "apple pie" but not "red apple"
- dog\$ matches "hot dog" but not "dogwood"

Pattern Cheat Sheet

Type	Pattern	Matches
Anchors	^ / \$	Start / End of line
Quantify	* / +	Zero+ / One+ repetitions
Groups	[] / ()	Sets / Capture groups
Wildcard	.	Any single character

Basic Examples

- ^apple matches "apple pie" but not "red apple"
- dog\$ matches "hot dog" but not "dogwood"
- ba+na matches "banana", "baana", but not "bana"

Pattern Cheat Sheet

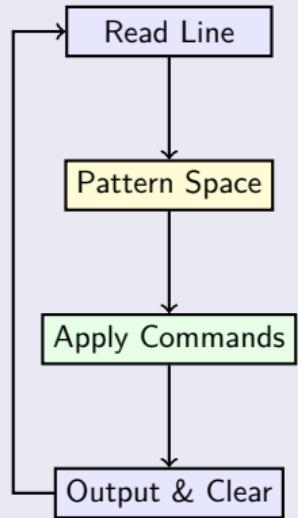
Type	Pattern	Matches
Anchors	^ / \$	Start / End of line
Quantify	* / +	Zero+ / One+ repetitions
Groups	[] / ()	Sets / Capture groups
Wildcard	.	Any single character

Basic Examples

- ^apple matches "apple pie" but not "red apple"
- dog\$ matches "hot dog" but not "dogwood"
- ba+na matches "banana", "baana", but not "bana"
- [0-9]{3} matches exactly three digits

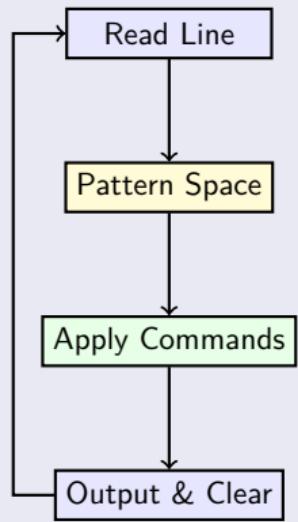
The sed Execution Model

The sed Cycle



The sed Execution Model

The sed Cycle



Key Concept

sed works line-by-line. It never modifies the original file unless you use the `-i` (in-place) flag.

sed: Search and Replace

Examples with Output

```
# Replace first 'foo' with 'bar'  
$ echo "foo foo" | sed 's/foo/bar/'  
bar foo
```

```
# Replace ALL 'foo' with 'bar' (global)  
$ echo "foo foo" | sed 's/foo/bar/g'  
bar bar
```

```
# Delete lines matching a pattern  
$ printf "line1\n#linecomment\nline2" | sed '/^#/d'  
line1  
line2
```

sed: Addressing and Ranges

Target Specific Lines

```
# Delete only the first 2 lines
```

```
$ sed '1,2d' file.txt
```

```
# Print only lines 5 through 10
```

```
$ sed -n '5,10p' file.txt
```

```
# Substitute only on lines containing 'error'
```

```
$ sed '/error/s/false/true/' log.txt
```

Practice: config cleanup

- 1 Task: Remove all comments (#) and empty lines from a file.

sed: Addressing and Ranges

Target Specific Lines

```
# Delete only the first 2 lines
```

```
$ sed '1,2d' file.txt
```

```
# Print only lines 5 through 10
```

```
$ sed -n '5,10p' file.txt
```

```
# Substitute only on lines containing 'error'
```

```
$ sed '/error/s/false/true/' log.txt
```

Practice: config cleanup

- 1 Task: Remove all comments (#) and empty lines from a file.

Answer

```
sed -e '/^#/d' -e '/^\$/d' config.txt
```

More sed Examples

Advanced Usage

```
# Change delimiter (useful for paths)
sed 's|/usr/local|/opt|g' paths.txt
```

```
# Backreferences (Swap first two words)
echo "Hello World" | sed 's/(\w+\ ) \(\w+\)/\2 \1/'
```

```
# Multiple commands
sed -e 's/foo/bar/g' -e 's/baz/qux/g' file.txt
```

```
# In-place editing (GNU/BSD differ; macOS needs -i '')
sed -i.bak 's/debug/info/g' config.yml
```

awk: The Pattern-Action Model

How it works

awk automatically loops over every line. You define **patterns** and **actions**.

```
{awk 'pattern { action }' file}
```

awk: The Pattern-Action Model

How it works

awk automatically loops over every line. You define **patterns** and **actions**.

```
{awk 'pattern { action }' file}
```

Basic Examples

```
# Print 1st and 3rd columns (default delimiter: space)
$ awk '{print $1, $3}' data.txt
```

```
# Filter by value: print line if 2nd col > 100
$ awk '$2 > 100' data.txt
```

```
# Use a custom delimiter (e.g., colon)
$ awk -F':' '{print $1}' /etc/passwd
```

awk: Context and Math

Built-in Variables

- **NR:** Number of Records (Line Number)
- **NF:** Number of Fields (Columns in current line)
- **\$0:** The entire line

awk: Context and Math

Built-in Variables

- **NR:** Number of Records (Line Number)
- **NF:** Number of Fields (Columns in current line)
- **\$0:** The entire line

Examples with Output

```
# Print line number and the line
$ awk '{print NR, $0}' file.txt
```

```
# Sum values in the first column
$ awk '{sum += $1} END {print "Total:", sum}' numbers.txt
Total: 450
```

```
# Calculate average of 1st column
$ awk '{s+=$1; c++} END {print "Avg:", s/c}' numbers.txt
```

Practice: Data Extraction

Challenge

You have a CSV (Name, Age, Score) called `class.csv`.

- ① Print only the names of students older than 20.
- ② Calculate the total number of words in a text file.

Practice: Data Extraction

Challenge

You have a CSV (Name, Age, Score) called `class.csv`.

- ① Print only the names of students older than 20.
- ② Calculate the total number of words in a text file.

Answers

- ① `awk -F',' '$2 > 20 {print $1}' class.csv`
- ② `awk '{total + NF}' END {print total}' file.txt=`

More awk Examples

Advanced Features

```
# Formatted printing (printf)
awk '{printf "Name: %-10s ID: %03d\n", $1, $2}' students.txt

# Length of string
awk 'length($0) > 80' file.txt # Print lines > 80 chars

# Logical conditions (AND/OR)
awk '$1 == "POST" && $9 == 200' access.log

# Range pattern (start, end)
awk '/START/,/END/' data.txt

# Filter by timestamp range and format output
awk '$1 >= "2026-01-20" && $1 <= "2026-01-25" \
{printf "[%s %s] %s\n", $1, $2, substr($0, index($0,$3))}' system.log
```

Scripting Basics

- Automate repetitive tasks
- Combine multiple commands
- Add logic (if/loops)
- Reproducibility

```
#!/bin/bash
# hello.sh
echo "Hello, $USER!"
echo "Date: $(date)"
```

Making it Executable

```
chmod +x hello.sh
./hello.sh
```

Variables in Scripts

Definition and Scope

```
# Definition: No spaces around =
name="STIC Class"
count=42
```

```
# Access: Use the $ sign
echo "Welcome to $name"
```

```
# Command Substitution: Use $()
current_user=$(whoami)
```

Important Rules

- **Quotes Matter:** Use "\$var" to prevent word splitting if your variable contains spaces.
- **Braces:** Use \${var} for clarity or when appending text: echo "\${name}_backup".

Positional Arguments

Communicating with Scripts

Variable	Description
\$0	The script name itself
\$1, \$2..	First, second arguments
\$#	Number of arguments passed
\$@	All arguments as a list
\$?	Exit status of the <code>last</code> command

Practice: Arg Grep

- ① Create a script that takes a word as \$1 and a file as \$2 and greps for it.
- ② **Answer:** `grep "$1" "$2"`
- ③ Use \$@ for looping over arguments.

Logic and Conditionals

```
if [ "$1" -gt 10 ]; then
    echo "Large"
elif [ -f "$1" ]; then
    echo "It's a file"
else
    echo "Small/Unknown"
fi
```

Test	True if...
-f	Regular file
-d	Directory
-eq	Equal (numeric)
-z	Empty string
=	Equal (string)

Practice: File Safety

- 1 Write a script that checks if a directory exists before creating it.

Logic and Conditionals

```
if [ "$1" -gt 10 ]; then
    echo "Large"
elif [ -f "$1" ]; then
    echo "It's a file"
else
    echo "Small/Unknown"
fi
```

Test	True if...
-f	Regular file
-d	Directory
-eq	Equal (numeric)
-z	Empty string
=	Equal (string)

Practice: File Safety

- ① Write a script that checks if a directory exists before creating it.
- ② **Answer:** `if [! -d "$dir"]; then mkdir "$dir"; fi`

Iteration (for, while)

```
# Range
for i in {1..5}; do
    echo "$i"
done
```

```
# Files
for f in *.txt; do
    mv "$f" "${f}.bak"
done
```

```
# Read file line-by-line
while read -r line; do
    echo "PROCESSED: $line"
done < file.txt
```

```
# Counter
while [ $c -lt 5 ]; do
    ((c++))
done
```

Practice: Bulk Rename

- ① Write a for loop that adds a .old extension to every file in the current directory.

Iteration (for, while)

```
# Range  
for i in {1..5}; do  
    echo "$i"  
done
```

```
# Files  
for f in *.txt; do  
    mv "$f" "${f}.bak"  
done
```

```
# Read file line-by-line  
while read -r line; do  
    echo "PROCESSED: $line"  
done < file.txt
```

```
# Counter  
while [ $c -lt 5 ]; do  
    ((c++))  
done
```

Practice: Bulk Rename

- ① Write a for loop that adds a .old extension to every file in the current directory.
- ② **Answer:** `for f in *; do mv "$f" "$f.old"; done`

Iteration (for, while)

```
# Range  
for i in {1..5}; do  
    echo "$i"  
done
```

```
# Files  
for f in *.txt; do  
    mv "$f" "${f}.bak"  
done
```

```
# Read file line-by-line  
while read -r line; do  
    echo "PROCESSED: $line"  
done < file.txt
```

```
# Counter  
while [ $c -lt 5 ]; do  
    ((c++))  
done
```

Practice: Bulk Rename

- ① Write a for loop that adds a .old extension to every file in the current directory.
- ② **Answer:** `for f in *; do mv "$f" "$f.old"; done`
- ③ `for f in *` is safer and faster than `for f in $(ls)`.

Iteration (for, while)

```
# Range  
for i in {1..5}; do  
    echo "$i"  
done
```

```
# Files  
for f in *.txt; do  
    mv "$f" "${f}.bak"  
done
```

```
# Read file line-by-line  
while read -r line; do  
    echo "PROCESSED: $line"  
done < file.txt
```

```
# Counter  
while [ $c -lt 5 ]; do  
    ((c++))  
done
```

Practice: Bulk Rename

- ① Write a for loop that adds a .old extension to every file in the current directory.
- ② **Answer:** `for f in *; do mv "$f" "$f.old"; done`
- ③ `for f in *` is safer and faster than `for f in $(ls)`.
- ④ Use while read for processing large files line-by-line.

Iteration (for, while)

```
# Range
for i in {1..5}; do
    echo "$i"
done
```

```
# Files
for f in *.txt; do
    mv "$f" "${f}.bak"
done
```

```
# Read file line-by-line
while read -r line; do
    echo "PROCESSED: $line"
done < file.txt
```

```
# Counter
while [ $c -lt 5 ]; do
    ((c++))
done
```

Practice: Bulk Rename

- ① Write a for loop that adds a .old extension to every file in the current directory.
- ② **Answer:** `for f in *; do mv "$f" "$f.old"; done`
- ③ `for f in *` is safer and faster than `for f in $(ls)`.
- ④ Use `while read` for processing large files line-by-line.
- ⑤ `((...))` is used for arithmetic in Bash.

Functions and Scope

```
greet() {  
    local name=$1  
    echo "Hello $name"  
}  
  
greet "Alice"
```

- Use **local** variables
- Return status (0-255)
- Arguments are \$1, \$2..
- Name functions clearly

Practice: Math Function

- ① Create a function square that prints the square of its first argument.

Functions and Scope

```
greet() {  
    local name=$1  
    echo "Hello $name"  
}  
  
greet "Alice"
```

- Use **local** variables
- Return status (0-255)
- Arguments are \$1, \$2..
- Name functions clearly

Practice: Math Function

- ① Create a function square that prints the square of its first argument.
- ② **Answer:** `square() { echo $(($1 * $1)); }`

Robust Scripting and Debugging

```
set -euo pipefail
```

- **-e**: Exit on error
- **-u**: Error on unset var
- **pipefail**: Pipeline fails if ANY part fails

- **set -x**: Print commands
- **bash -n**: Check syntax
- **shellcheck**: (External tool) highly recommended!

Practice: Safety First

- ➊ What happens if you run `rm -rf $DIR/file` and `$DIR` is undefined?

Robust Scripting and Debugging

```
set -euo pipefail
```

- **-e**: Exit on error
- **-u**: Error on unset var
- **pipefail**: Pipeline fails if ANY part fails

- **set -x**: Print commands
- **bash -n**: Check syntax
- **shellcheck**: (External tool) highly recommended!

Practice: Safety First

- ❶ What happens if you run `rm -rf $DIR/file` and `$DIR` is undefined?
- ❷ How does `set -u` prevent this disaster?

Robust Scripting and Debugging

```
set -euo pipefail
```

- **-e**: Exit on error
- **-u**: Error on unset var
- **pipefail**: Pipeline fails if ANY part fails

- **set -x**: Print commands
- **bash -n**: Check syntax
- **shellcheck**: (External tool) highly recommended!

Practice: Safety First

- ➊ What happens if you run `rm -rf $DIR/file` and `$DIR` is undefined?
- ➋ How does `set -u` prevent this disaster?
- ➌ **Answer:** Without `-u`, it runs `rm -rf /file`. With `-u`, the script exits immediately.

Example: Auto-Backup

A Robust Backup Script

```
#!/bin/bash
set -euo pipefail

DEST="/backups/$(date +%Y-%m-%d)"
mkdir -p "$DEST"

for dir in "$@"; do
    if [ -d "$dir" ]; then
        echo "Backing up $dir..."
        tar -czf "$DEST/$(basename "$dir").tar.gz" "$dir"
    else
        echo "Warning: $dir is not a directory" >&2
    fi
done
```

Analysis: Auto-Backup

Best Practices

- ① **Safety:** set -euo pipefail ensures the script stops if a directory can't be created or a command fails.
- ② **Dynamic:** Uses \$@ to process any number of folders passed as arguments.
- ③ **Error Handling:** Redirects warnings to stderr (>&2) so they don't pollute the standard output.
- ④ **Clean Paths:** Uses basename to ensure the archive name is clean even if a full path is provided.

Questions?

Thank you!

Ask your questions on **Piazza**