Lecture 1 Handout

Juni Kim

January 20, 2026

Contents

§1 Shell Script Execution Basics

This section covers execution details that are easy to miss in lecture but cause many confusing failures.

§1.1 Shebangs

A shell script is executed by the kernel, not by your shell. The first line (the **shebang**) determines which interpreter is used.

```
#!/usr/bin/env bash
```

Important points:

- The kernel reads the shebang and invokes the interpreter.
- `/usr/bin/env bash` is preferred over `/bin/bash`.
- `/bin/bash` is not guaranteed to exist on all systems.
- `/bin/sh` is the only interpreter required by POSIX.

If the shebang is missing or invalid, execution will fail even if the file exists.

§1.2 Executable Bit

A script file must be marked executable to be run directly:

```
# assuming script.sh is in your working directory
chmod +x script.sh
./script.sh
# this will fail unless your working directory is in $PATH
script.sh
```

Contrast with:

```
# again assumes script.sh is in your directory.
bash script.sh
```

Running with `bash script.sh` bypasses the shebang entirely. If a script only works this way, it may fail in grading or CI.

§1.3 `$0`, `dirname`, and `realpath`

Inside a script, `$0` expands to how the script was invoked, not necessarily its absolute path.

Common defensive pattern:

```
DIR="$(realpath "$(dirname "$0")")"
```

This allows scripts to:

- locate files relative to themselves
- be run from any working directory
- avoid reliance on caller state

§2 Variables and Quoting

This section covers variable behavior and quoting rules that frequently cause subtle bugs.

§2.1 Shell Variables vs Environment Variables

Shell variables are local to the shell. Environment variables are inherited by child processes.

```
FOO=hello         # shell variable
export BAR=world # environment variable
```

Only exported variables are visible to programs you run.

§2.2 Capturing Command Output

Use command substitution to capture stdout:

```
FILES="$(ls)"
FILECOUNT="$(ls | wc -l)"
```

Exit codes are **not** captured—only stdout.

§2.3 Quoting and Word Splitting

Unquoted variables are split on whitespace.

```
FILE="my file.txt"
rm $FILE       # WRONG: expands to two arguments
rm "$FILE"     # correct
```

Rule of thumb:

> Quote variables unless you explicitly want splitting.

§2.4 Globbing

Globs (`*`, `?`, `[...]`) expand **before** program execution.

```
echo *.txt
```

If no files match, the glob may remain unexpanded, depending on shell settings.

§3 Shell Misc I: Control Flow Footguns

This section covers control-flow behaviors that are often misunderstood.

§3.1 Exit Codes

Every command returns an exit code:

- `0` = success
- non-zero = failure

The shell variable `$?` contains the last exit code.

§3.2 `&&` and `||`

```
cmd1 && cmd2   # cmd2 runs only if cmd1 succeeds
cmd1 || cmd2   # cmd2 runs only if cmd1 fails
```

These operators are driven entirely by exit codes.

§3.3 `test`, `true`, and `false`

```
test -f file.txt
[ -d somedir ]
true
false
```

`test` and `[ ]` are just commands that return exit codes.
Negation inverts exit status:

```
! test -e file.txt
```

§4 Shell Misc II: Defensive Defaults

This section covers defaults and patterns expected in this course.

§4.1 `set -euo pipefail`

Recommended at the top of scripts:

```
set -euo pipefail
```

Effects:

- `-e`: exit on unhandled error
- `-u`: error on use of unset variables
- `pipefail`: pipelines fail if any command fails

Caveats:

- `-e` does not trigger inside all conditionals
- `-o pipefail` is a bash-only feature (fails on busybox sh).
- control-flow semantics change subtly
- this is not a substitute for careful reasoning

§4.2 Idempotent Filesystem Operations

Filesystem state persists across runs. Scripts must tolerate re-execution. In particular, if you did not create it in the script, **do not assume it exists**.

Prefer:

```
mkdir -p output
rm -f temp.txt
```

Be deliberate about:

- > vs >>
- overwriting vs appending
- cleanup on failure

§5 Docker

Docker is used to standardize the **host system**. It does not eliminate the host/target distinction.

§5.1 Docker Images

An image is a filesystem snapshot plus metadata.

Key ideas:

- Images are built from Dockerfiles.
- Images are immutable once built.
- Building an image does not run your program.

```
docker build -t myimage .
```

§5.2 Docker Containers and Volumes

A container is a running instance of an image.

```
docker run -it myimage
```

Volumes and bind mounts allow sharing directories with the host:

```
docker run -it -v "$PWD:/work" myimage
```

Important:

- Mounted directories are shared mutable state.
- Files created in the container may persist on the host.
- Deleting a container does not delete volumes.

For debugging:

- Containers can be stopped and restarted.
- Images must be rebuilt to change their contents.

§5.3 Inspecting and Managing Containers

Docker keeps track of containers independently of your shell. A very common source of confusion is forgetting which containers are running.

§5.3.1 `docker ps`

The command `docker ps` lists **currently running containers**:

```
$ docker ps
CONTAINER ID   IMAGE         COMMAND                    CREATED         STATUS
     PORTS     NAMES
a3c1f4b92d3e   debian:12     "/bin/bash"                3 minutes ago   Up 3 minutes
               hopeful_morse
```

Important columns:

- `CONTAINER ID`: unique identifier (shortened hash)
- `IMAGE`: image the container was created from

- `COMMAND`: entrypoint / command being run
- `STATUS`: whether the container is running
- `NAMES`: human-readable name (auto-generated if not specified)

To see **all containers**, including stopped ones:

```
docker ps -a
```

Stopped containers still exist and still consume metadata until removed.

§5.3.2 Stopping Containers

To stop a running container:

```
docker stop hopeful_morse
```

You may also use the container ID:

```
docker stop a3c1f4b92d3e
```

Stopping a container sends a termination signal to the process inside.

§5.3.3 Removing Containers

To permanently delete a container:

```
docker rm hopeful_morse
```

Only stopped containers can be removed. To stop and remove in one step:

```
docker rm -f hopeful_morse
```

This is often useful during development.

§5.3.4 Common Pitfalls

- Exiting a shell inside a container does not delete the container.
- Containers can accumulate if not explicitly removed.
- Volumes and bind mounts persist even after containers are deleted.