ECS 150 - Project 1

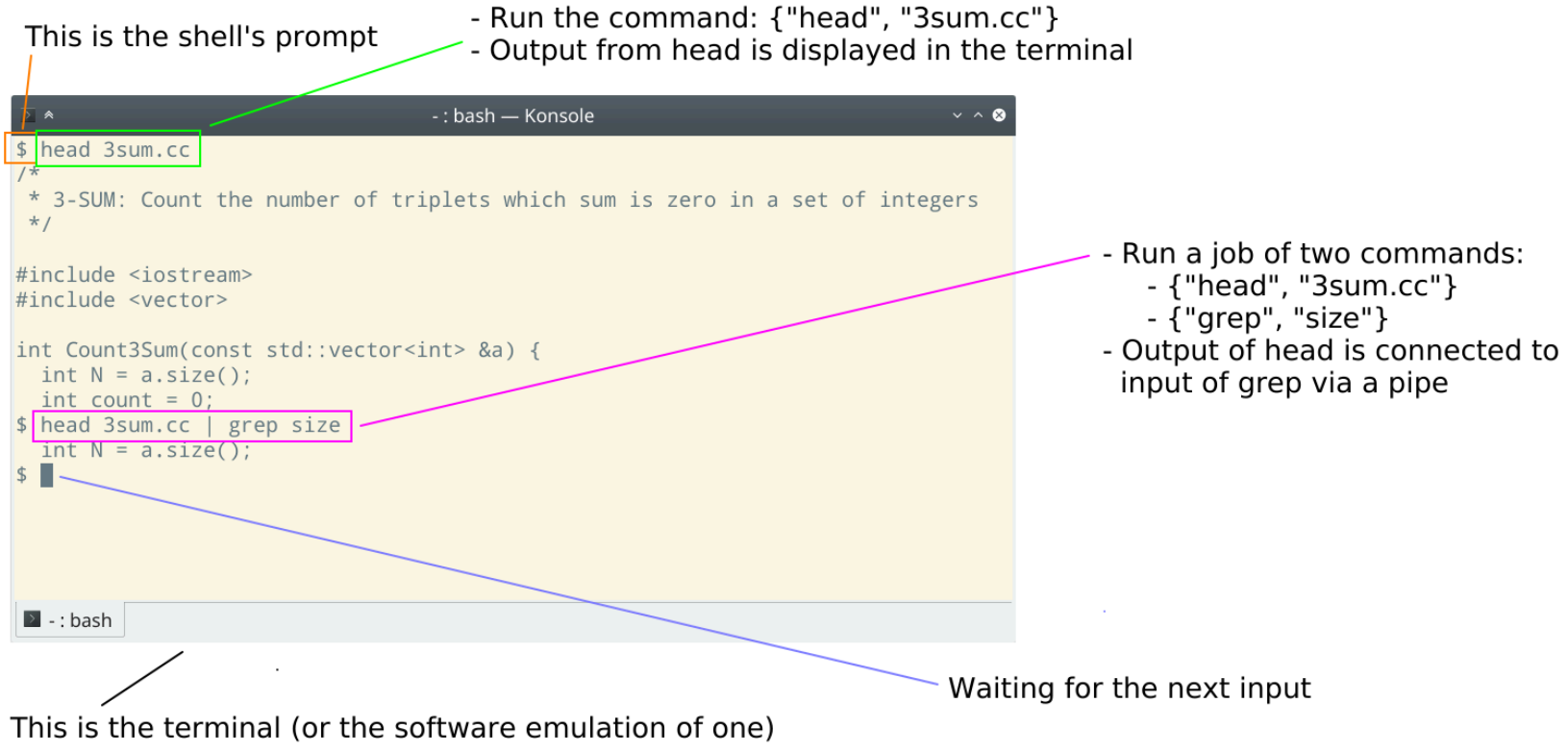
Prof. Joël Porquet-Lupine

UC Davis - SQ25

Shell, an introduction

What's a shell?

- User interface to the Operating System's services
- Gets input from user, interpret the input, and launch the desired action(s)



Shell, an introduction

Some big names

Name	Comment	First released
Thompson shell	First Unix shell	1971
Bourne shell	Default shell for Unix 7	1977
Bash	Default on most Linux distributions	1989
Zsh	My favorite shell :D (now default on MacOS!)	1990
Fish	New(ish) kid in town -- tries to be more user friendly than other shells	2005

- Big and old pieces of software
- Bash: 35 years old and ~200,000 lines of code!

Simple shell

Goal

- Understand important UNIX system calls
- Implementing a simple shell called **sshell**

Specifications Part 1 -- Solo

- Execute commands with arguments

```
sshell@ucd$ date -u
```

Specifications Part 2 -- Group (later)

- Redirect standard output of command to file

```
sshell@ucd$ date -u > file
```

- Pipe the output of commands to other commands

```
sshell@ucd$ cat /etc/passwd | grep root
```

- Offer a selection of builtin commands

```
sshell@ucd$ cd directory
sshell@ucd$ pwd
/home/jporquet/directory
```

- Two extra features (TBD)

Simple shell

Commands with arguments

1. Display prompt
2. Read command from input
   - Potentially composed of multiple arguments (up to 16 total)
3. Execute command
4. Wait for completion
5. Display information message

```
sshell@ucd$ echo Hello world
Hello world
+ completed 'echo Hello world' [0]
sshell@ucd$ sleep 5
+ completed 'sleep 5' [0]
```

Builtin commands

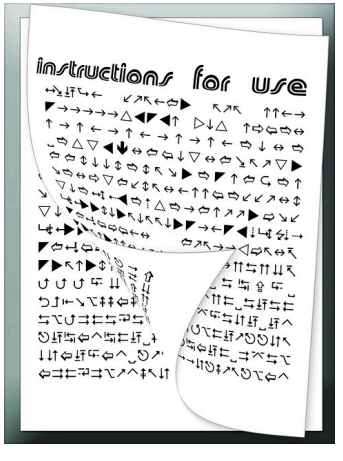
- Most commands are provided by external executables
  - `/usr/bin/echo`, `/usr/bin/ls`, etc.
- Some commands have to be provided by the shell directory
  - only `exit` for Part 1 (more for Part 2)

```
sshell@ucd$ exit
Bye...
+ completed 'exit' [0]
```

General information

Project assignment

- Project assignment published this morning!
- Read assignments multiple times and follow the instructions
  - Work phases to help with your progression
- Stay up-to-date
  - Extra information given during lecture or discussion
  - Class announcements
  - Piazza



General information

Deadline

- Project is due by **Thursday, April 20th, 2025, at 11:59pm**
- No extension will be given
- Progressive late penalty
  - $\frac{(1.3471^x - 1)}{(1.3471 - 1)}$ where x is decimal number of hours late
  - Only -2.3% for 2h late!
  - Then -14.3% for 6h late, -53.8% for 10h late, and -100% for 12h late.

- Don't wait, both parts of this project are usually considered to be intense!

General information

Academic integrity

On your end

- Projects are to be written **from scratch**
  - Even if you already took (part of) this class
- Projects are to be written **in equal proportion** by both partners
- Avoid using snippets of code you find online (e.g., stackoverflow)
  - Instead rewrite them yourself
  - Cite your sources
- ChatGPT and such are not authorized
  - (Explanation of concepts OK, but code generation is not)

On my end

- Use of MOSS on **all submissions** and comparison with previous quarters
- If you find existing source code available online (e.g., Github)
  - Will definitely appear via MOSS!
- Transfer all misconduct cases to SJA
  - At best, fail the project
  - At worst, fail the class (and even get suspended or dismissed from the university if not first offense)

Makefile

Intro

- A `Makefile` is a file containing a set of rules
  - Represents the various steps to follow in order to build a program
  - Building recipe
- Used with the build automation tool `make`

Anatomy of a rule

```
target: [list of prerequisites]
[ <tab> command ]
```

- For `target` to be generated, the prerequisites must all exists (or be themselves generated if necessary)
- `target` is generated by executing the specified command
- `target` is generated only if it does not exist, or if one of the prerequisites is more recent
  - Prevents from building everything each time, but only what is necessary

Makefile

Simple `Makefile`

```
$ ls
Makefile prog.c utils.c utils.h
```

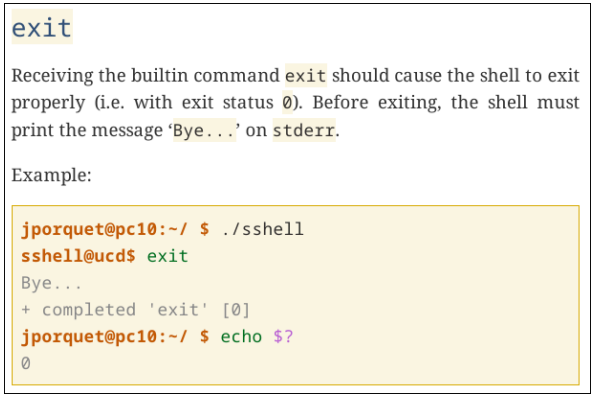
```
# Generate executable
myprog: prog.o utils.o
    gcc -Wall -Wextra -Werror -o myprog prog.o utils.o

# Generate objects files from C files
prog.o: prog.c utils.h
    gcc -Wall -Wextra -Werror -c -o prog.o prog.c
utils.o: utils.c utils.h
    gcc -Wall -Wextra -Werror -c -o utils.o utils.c

# Clean generated files
clean:
    rm -f myprog prog.o utils.o
```

- Adapt this code to your needs
  - (Could be actually a lot simpler than this: intermediate object generation is not necessary if only one C file)

Automatic grading

Overview

- Project fully autograded
- Aims to capture **correctness**
  - Does your code implement the given specifications?



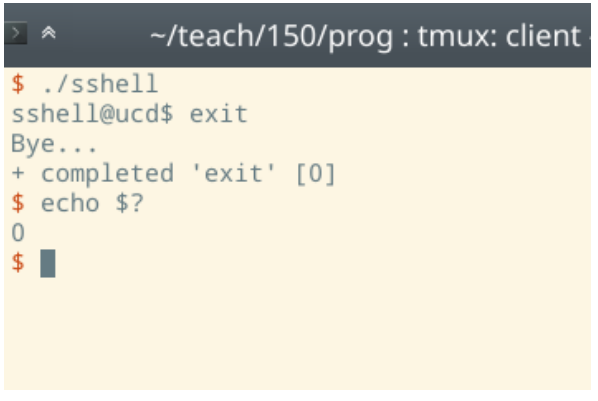
`exit`

Receiving the builtin command `exit` should cause the shell to exit properly (i.e. with exit status `0`). Before exiting, the shell must print the message '`Bye...`' on `stderr`.

Example:

```
jporquet@pc10:~/ $ ./sshell
sshell@ucd$ exit
Bye...
+ completed 'exit' [0]
jporquet@pc10:~/ $ echo $?
0
```

Specs



Execution

Testing script

- Complete grading test script has more than 10 test cases
  - Covers all the features, with multiple scenarios, error management, etc.
- Partial test script will be published early next week
  - 4 test cases, one per main feature

Automatic grading

Manual test cases

- All the test cases can be reproduced manually

```
cmd_no_arg_1() {
    ...
    touch titi toto
    run_test_case "ls\nexit\n"
    ...
    local line_array=()
    line_array+=("$(select_line "${STDOUT}" "2")")
    line_array+=("$(select_line "${STDOUT}" "3")")
    local corr_array=()
    corr_array+=("titi")
    corr_array+=("toto")
    ...
}
```

Script

```
$ mkdir test && cd test
$ touch titi toto
$ echo -e "ls\nexit\n" | ../sshell
sshell@ucd$ ls
titi  toto
+ completed 'ls' [0]
sshell@ucd$ exit
Bye...
$ echo -e "ls\nexit\n" | ../sshell 2>/dev/null
sshell@ucd$ ls
titi  toto
sshell@ucd$ exit
$
```

Manual test case

Advice

- Make sure that you pass the given test script
  - Test as soon as the script is published next week!
- Re-read the assignment prompt entire and carefully
  - Reproduce **all** the examples shown
  - Especially regarding error handling...

Best practices

Implementation quality

Rationale

- Writing some code that implements certain specs is not enough
- A good code is easy to understand, manipulate, and extend

Data structures and code refactoring

- Use the right data structures
  - If you're using `char ***` in your code, that's probably a subpar approach
- Split your functions the right way
  - Ideally, one function per logical functionality

Code design

- Don't over-complicate your design/code
  - *Simple is always the best option*
- Don't be scared to rewrite big chunks of your code at some point
  - That's how any large project works!

Best practices

Coding style

- Keep it consistent
  - Don't mix tab and spaces
  - Keep the same indentation (typically at least 4)
- Name your variable properly
  - Example of poor variable names:

```
int a, b, tmp;
```

  - Better variable names:

```
int mypipe[2], infile,
outfile;
```

- Comment your code (with meaningful comments)
  - Example of poor comment:

```
i++; // increment variable i
```

  - Better comment:

```
// Skip heading whitespaces
while (*c && isspace(*c))
    c++;
```

In preparation for Part 2...

Group work

- Teams of exactly two partners
  - Find a partner after/before class or using "Course Chat" on CourseAssist
- Find a partner with whom you can work well
  - Define what kind of collaboration you're looking for before pairing up
  - How to meet? How regularly? Etc.
- Look into effective group programming approaches
  - E.g., pair programming