

ECS 150 - Project 1 - Part I

Prof. Joël Porquet-Lupine

UC Davis - 2020/2021



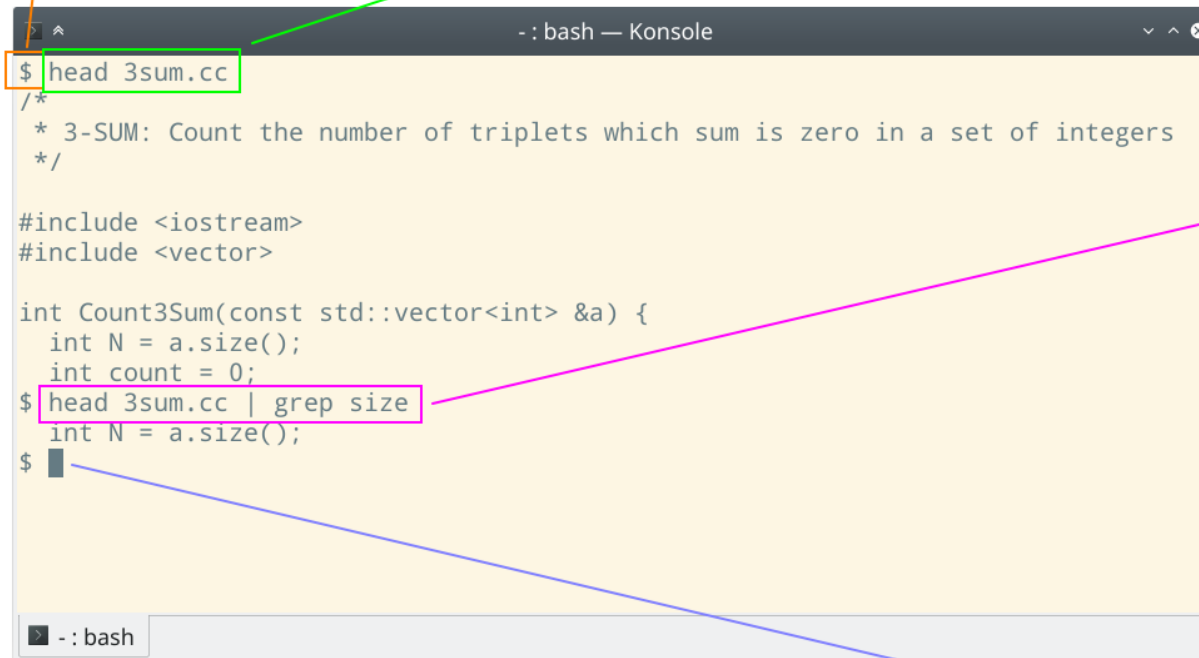
Shell, an introduction

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)

This is the shell's prompt

- Run the command: {"head", "3sum.cc"}
- Output from head is displayed in the terminal



A terminal window titled "- : bash — Konsole" showing the contents of a file named 3sum.cc. The file contains C++ code for counting triplets with a sum of zero. The terminal prompt is "\$". The command "head 3sum.cc" has been entered and is highlighted with a green box. Below it, the command "head 3sum.cc | grep size" is entered and highlighted with a pink box. The terminal is waiting for the next input, indicated by a blue line pointing to the prompt.

```

$ head 3sum.cc
/*
 * 3-SUM: Count the number of triplets which sum is zero in a set of integers
 */

#include <iostream>
#include <vector>

int Count3Sum(const std::vector<int> &a) {
    int N = a.size();
    int count = 0;
    head 3sum.cc | grep size
    int N = a.size();
$
```

- Run a job of two commands:
 - {"head", "3sum.cc"}
 - {"grep", "size"}
- Output of head is connected to input of grep via a pipe

This is the terminal (or the software emulation of one)

Waiting for the next input

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 kid in town -- tries to be more user friendly than other shells	2005

- Big and old pieces of software
- Bash: 30 years and ~200,000 lines of code!

Simple shell

Goal

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

Specifications

- Execute commands with arguments

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

- 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
```

- And some extra features...

Simple shell

Commands

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

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

Simple shell

Builtin commands

```
sshell@ucd$ pwd
/home/jporquet/ecs150/
+ completed 'pwd' [0]
sshell@ucd$ cd ..
+ completed 'cd ..' [0]
sshell@ucd$ pwd
/home/jporquet/
+ completed 'pwd' [0]
sshell@ucd$ exit
Bye...
+ completed 'exit' [0]
```

- Most commands are provided by external executables
 - `/bin/echo`, `/bin/ls`, etc.
- Some commands have to be provided by the shell directory
 - `cd`, `exit`, etc.

Simple shell

Standard output redirection: >

```
sshell@ucd$ echo Hello world>file
+ completed 'echo Hello world>file' [0]
sshell@ucd$ cat file
Hello world
+ completed 'cat file' [0]
```

- Output redirection means that the process's output will be written to a file instead of to the terminal
- Spacing shouldn't matter
 - `echo Hello world>file` is equivalent to `echo Hello world > file`

Simple shell

Pipeline of commands: |

```
sshell@ucd$ echo Hello world | grep Hello|wc -l
1
+ completed 'echo Hello world | grep Hello|wc -l' [0][0][0]
sshell@ucd$
```

- Interconnection of multiple commands into a *job*
- Output of command before '|' is redirected as the input of the command located right after
- Up to three pipes on the same command line

Simple shell

Extra feature #1: Output redirection appending

```
sshell@ucd$ echo Hello > output
+ completed 'echo Hello > output' [0]
sshell@ucd$ echo world >> output
+ completed 'echo world >> output' [0]
sshell@ucd$ cat output
Hello
world
+ completed 'cat output' [0]
```

- `>>` appends to output file
- `>` truncates output file

Simple shell

Extra feature #2: `ls`-like builtin command

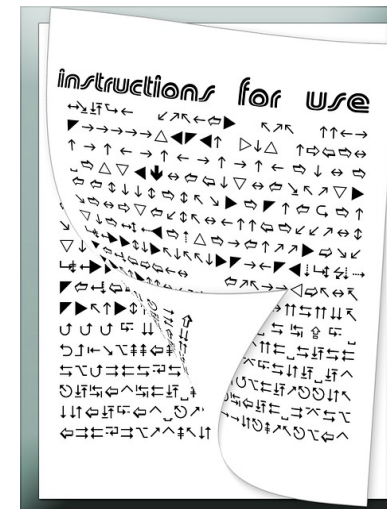
```
sshell@ucd$ pwd
/  
+ completed 'pwd' [0]  
sshell@ucd$ sls  
lib (7 bytes)  
home (4096 bytes)  
srv (4096 bytes)  
lost+found (16384 bytes)  
etc (4096 bytes)  
opt (4096 bytes)  
var (4096 bytes)  
lib64 (7 bytes)  
...
```

- `sls` to print contents of current directory

General information

Project assignment

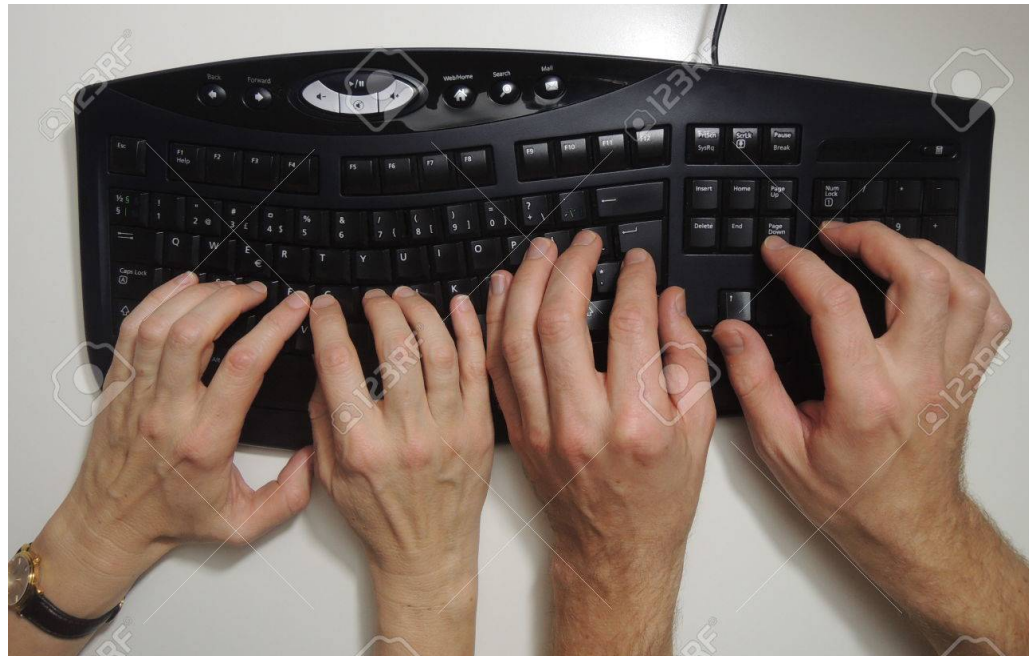
- Project assignment was published on Monday
- Read assignments multiple times and follow the instructions
 - Suggested phases to help with your progression
 - Recommended to follow but you don't have to
- Stay up-to-date
 - Extra information given during lecture or discussion
 - Class announcements
 - Piazza



General information

Group work

- Teams of exactly **two partners**
 - Find a partner on Piazza ([@5](#))
- 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.



General information

Deadline

- Project is due by **Thursday, October 22nd, 2020, at 11:59pm**
- **No extension will be given**
- 5% penalty for each hour late
 - Prorated by minutes
- Start early, this project is 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

On my end

- Use of MOSS on all submissions and comparison with previous quarters
- If you find existing source code available online
 - Will most likely appear via MOSS!
- Transfer to SJA in case of suspected misconduct
 - At best, fail the project
 - At worst, fail the class (and even get suspended or dismissed if not first offense)

Git

Introduction

Version control system for tracking changes in computer files and coordinating work on those files among multiple people.

Unlimited private repositories on [github](#) and [gitlab](#).

Initial configuration

```
$ git config --global user.name "Firstname Lastname"  
$ git config --global user.email "name@ucdavis.edu"
```

Git

How to start?

1. Create account and **private** repository online
2. Add partner as collaborator
3. Clone it locally

```
$ git clone git@github.com:nickname/ecs150-sshell.git && cd ecs150-sshell
```

4. Start coding

```
$ vim sshell.c
```

5. Commit and push

```
$ git add sshell.c  
$ git commit -m "Initial commit"  
$ git push
```

6. Your partner can now pull your commit

```
$ git pull
```

More resources

- <https://guides.github.com/activities/hello-world/>
- <https://www.atlassian.com/git/tutorials>

Git

Workflow

- Commit often
- One logical change per commit
- Write meaningful commit messages
- Study and understand what your partner commits

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 exist (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

```
# 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)

Best practices

Implementation quality

- Writing some code that implements certain specs is not enough
- A good code is easy to extend
- Use the right data structures
 - If you're using `char ***` in your code, that's probably the wrong approach
- Split your functions the right way
 - Ideally, one function per logical functionality
- 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

- Consistent
 - Don't mix tab and spaces
 - Keep the same indentation (at least 4)
- Comment your code (with meaningful comments)
 - Example of poor comment: `i++; // increment variable i`
- Name your variable properly:
 - Example of poor variable name: `int temp;`
- Remove *dead* code
- If you want to become a pro, start acting like one now:
 - Don't submit a draft!
 - Submit a program that works and is nice to read