ECS 150 - Project 1

Prof. Joël Porquet-Lupine

UC Davis - 2020/2021

UC DAVIS
COMPUTER SCIENCE

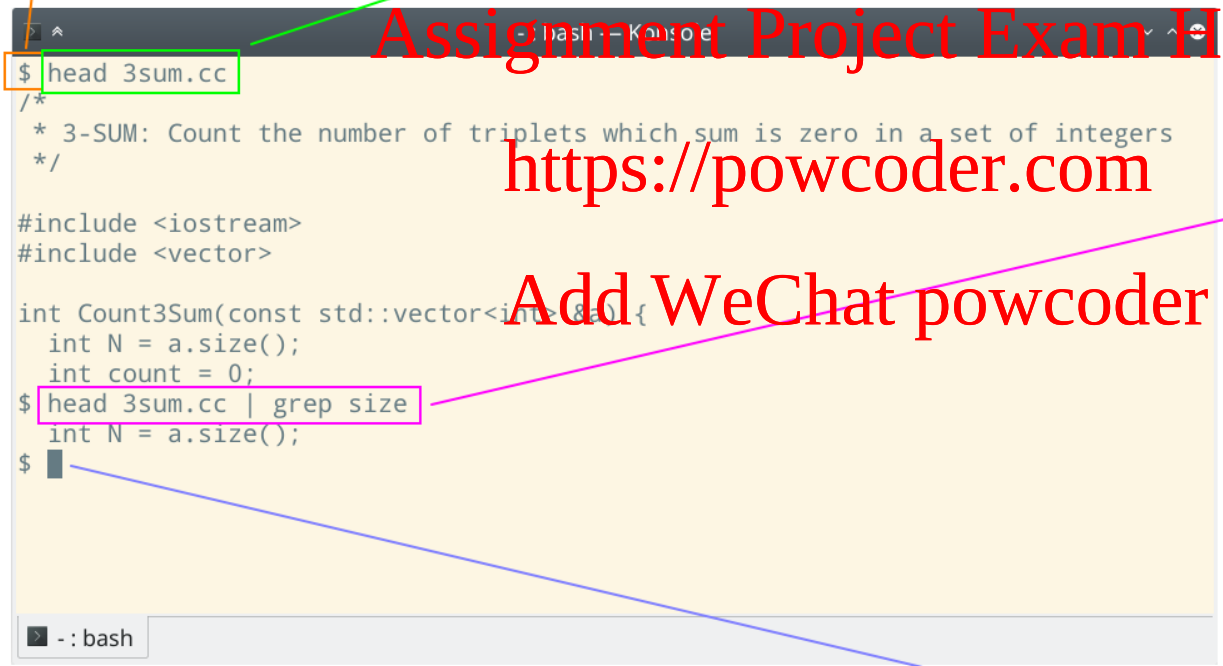
Shell, an introduction

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



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

Waiting for the next input

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

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 feature(s)...

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]
```

- 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: variables

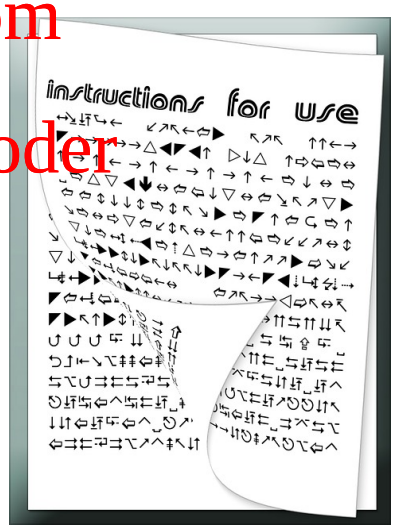
```
sshell@ucd$ set j toto
+ completed 'set j toto' [0]
sshell@ucd$ set i $j
+ completed 'set i $j' [0]
sshell@ucd$ echo $i
toto
+ completed 'echo $i' [0]
```


- 26 variables, from `a` to `z`
- `set` to initialize or reset a variable
- `$` to use variable

General information

Project assignment

- Project assignment was published this morning
- 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 (post `@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 **Friday, January 22nd, 2021, 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 (e.g., Github)
  - 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 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`

```
# 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 understand, manipulate, and 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 your 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

ECS 150 - Project 1

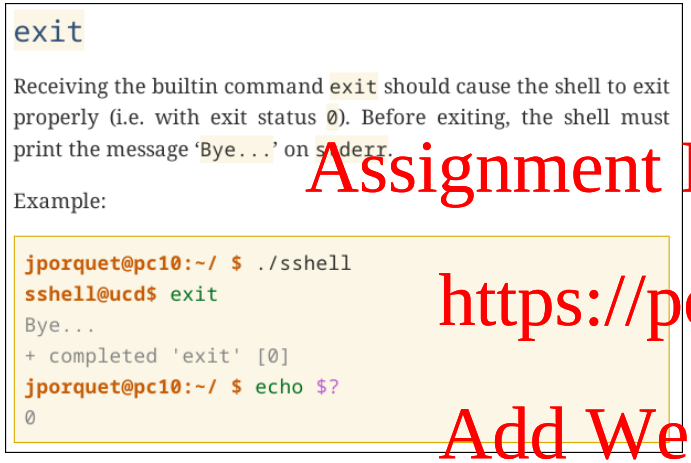
Prof. Joël Porquet-Lupine

UC Davis - 2020/2021

Automatic grading

Overview

- Automatic grading is ~60%
- (Partially) captures **correctness**
  - Does your code implement the given specifications?



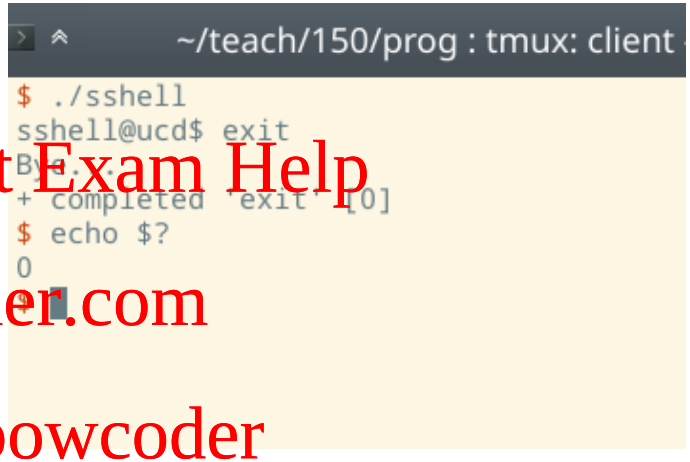
`exit`

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



```
~/teach/150/prog : tmux: client
$ ./sshell
sshell@ucd$ exit
Bye...
+ completed 'exit' [0]
$ echo $?
0
```

Execution

Testing script

- Complete grading test script has more than 20 test cases
  - Covers all the features, with multiple scenarios, error management, etc.
- Partial test script will be published later today
  - 7 test cases, one per main feature

Automatic grading

Manual test cases

- All the test cases can be reproduced manually

```bash
run_cmd_no_arg() {
    ...
    touch titi toto
    sshell_test "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
  - Do it right away, before you continue adding any new features!
  - Don't assume that the grading test script will work differently
- Re-read the assignment prompt entire and carefully
  - Reproduce **all** the examples shown
  - Especially regarding error handling...

Manual review

Overview

- Manual review is ~40%
- (Partially) captures **quality of project**
- Various aspects related to real-life programming projects
  - Is the project well-packaged?
  - Is the code properly designed and implemented?
  - Is the code well-explained?
  - Does the code follow good coding style?

Score breakdown

Easy points (~30%)

- Submission : ~10%
- Makefile: ~10%
- Code style: ~10%

Harder points (~70%)

- Report file: ~40%
- Quality of implementation: ~30%

Manual review

Submission

AUTHORS.csv

- File containing information about partners

Compilation

- Successful compilation of project when typing `make`

Contents

- Include only what's necessary, nothing more
  - Source code (`*.c`, `*.h`)
  - Makefile
  - Report file
  - *Optionally:* custom testing script or other relevant files
  - (Git specific files such as .gitignore are fine)

- Example of clutter that will affect your submission grade
  - Core dumps, backup files, object or executable files, hidden macOS folders (`.__MACOSX`, `.DS_Store`), etc.

Manual review

Makefile

- gcc with `-Wall -Wextra -Werror`
  - And compile without any warnings or errors
- Properly constructed rules
  - No use of implicit rules
  - Simple rules whenever possible
- Inclusion of functional `clean` rule

Manual review

Report

Rationale

- Ability to explain code from a high-level point of view
- Important for many aspects of software development
  - In-team communication
  - Out-group communication
  - Problem solving
  - Help guides and documentation
  - Software design and empathy
  - Career value
- https://www.computer.org/publications/tech-news/trends/why-every-developer-should-know-a-bit-of-technical-writing

Manual review

Report

Dos and dont's

- Explain how you transformed the assignment into code
  - Not "What does your code do?"
  - But "How does it do it?"
- Focus on how your program works at the time you submit it
  - Don't go too much into the details of everything you tried to get there
- Mention the limitations if any
- Don't repeat the assignment
- Don't write a book and don't explain every line of code...
- See example documentation published for Project #0

Important reminder

- The report file is 40% of the manual review grade (so ~15% of the total)
- Take care of the report instead of wasting time adding another feature at the last minute!

Manual review

Report

Formatting

- Name of the file is `REPORT.md` (not `README.md`, nor `REPORT.txt`, etc.)
- Formatted in Markdown
  - Specifications: https://guides.github.com/features/mastering-markdown/
  - Nothing fancy necessary, but at least: Headers, Some text formatting (italic, bold), Lists, Code blocks
  - Formatted in lines of **80 characters** maximum


```
# Specifications

*Note that the specifications for this project are subject to change at any time
for additional clarification. Make sure to always refer to the latest version.*

## Introduction

The goal of this project is to understand important UNIX system calls by
implementing a simple shell called **sshell**. A shell is a command line
interpreter: it accepts input from the user under the form of command line and
executes them.

In the following example, it is the shell that is in charge of printing the
*shell prompt*, understanding the supplied command line (redirect the output of
executable program `ls` with the argument `-l` to the input of executable
program `cat`), execute it and wait for it to finish before prompting the user
for a new command line.

```console
jporquet@pc10:~/ecs150 % ls -l | cat
total 12K
-rw------- 1 jporquet users 11K 2017-04-04 11:27 ASSIGNMENT.md
jporquet@pc10:~/ecs150 %
```

Similar to well-known shells such as *bash* or *zsh*, your shell will be able
to:

1. execute user-supplied commands with optional arguments
1. offer a selection of builtin commands
```

Good formatting :)

```
Functions:
NONCANNONICAL: contains functions provided by the instructor for implementing non-canonical input mode and our function rea
»d the user's input and decides what to do.

<CTRL-D> break the loop and ends the process if no children are running "exit" breaks the loops, sends "bye.." to err outp
»s.  <DELETE_KEY> replaces last symbol on screen with blank <ENTER_KEY> prints new line, pushes command line to History, p
»inished.

History is an array of 10 elements with a head pointer history_head, a current pointer history_pointer and a history_count
»pointer points to the current element while moving through the history. Pushing to history always writes to head element
»s over the oldest elements so it doesn't moves the current pointer to the previous slot and replaces the current command wit
»h place. After reaching oldest elements makes a bell sound. Down arrow moves the the pointer to the next element and rep
»ommand.

SHELL_FUN.H: header file for shell_fun.c

SHELL_FUN.C: contains all the other functions to run the shell.

-Helper functions: deal with c strings. write, copy, compare, print messages, search for symbols.

-Parse functions:
--parse_input_line(): takes a c string and breaks it into tokens for execvp to read. Outputs a array of tokens. Adds symbo
»oid parsing again.
--split_cmd(): splits an array of tokens by the '|' symbol for piping. Returns an array of token arrays to feed into pipe.

-Internal Commands:
--EXIT, CD and PWD are handled by the shell itself. A child process calling exit or cd wouldn't affect the parent, hence t
```

Poor formatting :(

Manual review

Coding style

- Go through your code starting from the beginning
- Reformat all the code with the same indentation (spaces vs tabs)
- Rename variables or functions that are poorly named
- Space out your code so that it looks nice to read
- Etc.

Quality of implementation

- Think about your overall design
  - Make it as simple and straightforward as possible
- Use of proper data structures

  - Don't over use arrays, they're not the solution to everything
- Split code into appropriate functions
- Avoid memory leaks

Manual review

Submit production code

```c
int main( void)     {
  func1(a, b  , c);
       func2(a,b,c);
    // printf("Back from func2\n");
    // Still need to investigate func3()
     func3();
    /* Also need to make a haircut appt */

            // Return value 0
    return 0;
 }
```

```c
int main(void)
{
    /* Parse the command line */
    func1(a, b, c);
    func2(a, b, c);

    /* Run requested command(s) */
    func3();

    return 0;
}
```

```makefile
myexec: mycode.c
    gcc -g -Wall -Werror ...
```

```makefile
myexec: mycode.c
    gcc -O2 -Wall -Werror ...
```


- What happens in development "stays" in development
- Production code (i.e., your submission) is a final product, not a draft
- Remove *dead code* and debugging comments
- Compile for performance, not debug

Coding tips

Fixed values (1)

```c
int main(void)
{
    char cmdline[512];

    fgets(cmdline, 512, stdin);
    ...

    return 0;
}
```

```c
#define CMDLINE_MAX 512

int main(void)
{
    char cmdline[CMDLINE_MAX];

    fgets(cmdline, CMDLINE_MAX, stdin);
    ...

    return 0;
}
```


- Avoid hardcoded values and prefer named macros
- Definition in one location, easy to update

Coding tips

Fixed values (2)

```c
void error_message(int error_code)
{
    switch(error_code) {
        case 0:
            fprintf(stderr,
                    "Error: missing command\n");
            break;
        case 1:
            fprintf(stderr,
                    "Error: command not found\n");
            break;
        case 2:
            ...
    }
}

void func(void)
{
    ...
    error_message(1);
    ...
}
```

```c
enum {
    ERR_MISSING_CMD,
    ERR_CMD_NOTFOUND,
    ...
};

void error_message(int error_code)
{
    switch(error_code) {
        case ERR_MISSING_CMD:
            fprintf(stderr,
                    "Error: missing command\n");
            break;
        case ERR_CMD_NOTFOUND:
            fprintf(stderr,
                    "Error: command not found\n");
            break;
        ...
    }
}

void func(void)
{
    ...
    error_message(ERR_CMD_NOTFOUND);
    ...
}
```

- Again, avoid hardcoded values
- Use generic constructs

Coding tips

Error checking

```c
int myfunc(char *buffer, int len)
{
    int i;

    if (buffer) {
        if (len >= 0) {

            /* Process buffer */
            for (i = 0; i < len; i++)
                ...

            return 0;
        }
    }

    return -1;
}
```

```c
int myfunc(char *buffer, int len)
{
    int i;

    /* Error checking */
    if (!buffer || len < 0)
        return -1;

    /* Process buffer */
    for (i = 0; i < len; i++)
        ...

    return 0;
}
```


- Avoid deep nesting of code
- Process error cases first, then proceed with actual code, at the same level

Coding tips

Function order

```c
#include <stdio.h>

int func1();
char func2(int);
int func3(char);

int main(void)
{
    func3();
}

int func1()
{
    func2();
    func3();
}

char func2(int)
{
    func3();
}

int func3(char)
{
}
```

```c
#include <stdio.h>

int func3(char)
{
}

char func2(int)
{
    func3();
}

int func1(void)
{
    func2();
    func3();
}

int main(void)
{
    func3();
}
```


- Popular design pattern in big C project, such as the Linux kernel
- Less code, less prototypes to maintain
- With a proper code editor function navigation is easy
  - Look into `ctags` or `cscope` (and interface with `vim` or `emacs`)

Coding tips

Functions

```c
#include <stdio.h>

int main(void)
{
    /* 1. First we do this */
    ...
    /* 2. Then we do that */
    ...
    /* 3. Then it depends */
    if (value == 42) {
        /* 3a. Some code using 42 */
    } else {
        /* 3b. Same code not using 42 */
    }

    return 0;
}
```

```c
#include <stdio.h>

int func1(char)
{
}

char func2(int)
{
}

int func3(int param)
{
}

int main(void)
{
    func1();
    func2();
    func3(value);
}
```


- Split your code into tinier chunks
  - If a function starts doing multiple things, time to break it down
- When possible, write generic and parameterizable functions

Coding style tips

Editor configuration

What you might see in your code editor:

```
struct linked_list {

  struct linked_list_node *head;
  int size;

};
```

What I see in mine:

```
struct linked_list {

  struct linked_list_node *head;
        int size;

};
```


- Ensure visual consistency between code editors
  - Spaces vs tabs
  - Settle for one convention with your partner, and configure your editors accordingly
- Remove unnecessary spaces
  - Take actual space in the source code