Software Development

Solving Homework Challenges Like a Professional Software Developer

C Programming Preparation

Programming References

Resource	Why use it?
Man pages! See section 3 for a lot of library functions. E.g. *man 3 printf*	● The manual is **at your fingertips** without leaving your terminal ○ Many text editors have shortcuts to open the man page for a function you have selected ● Answers *how* to use a function or feature and *what* it does. Often includes examples.
CPP Reference: https://en.cppreference.com/w/c/language	● It is **searchable**! ● Use it as a **reference** document ○ Which functions exist in a category? ○ How do I use this function? Can I use it in my version of C?
Brian Kernighan and Dennis Ritchie *The C Programming Language* https://archive.org/details/cprogramminglang00denn (also known as K&R)	● Familiar format for **textbook** learners ● Similar content to these slides, but more **detailed** ● It is a **well-known** book with programmers, so it is at least worth skimming so you can join a conversation
GDB quick reference https://users.ece.utexas.edu/~adnan/gdb-refcard.pdf	● You will need gdb a lot when debugging C projects ● This 2-page reference card has everything you need for day-to-day **debugging**, and it is **searchable**!

This slide is adapted from Prof. Egele's Boston University EC440 lectures

What to Review

If you haven’t recently programmed with C, please review:

- Functions: declarations and definitions
- Variables: definitions, usage
- Data types and conversion between them
- What operators are available
- Control flow (if statements, loops)
- Arrays and strings, array-to-pointer decay
- Pointers and pointer arithmetic
- Memory management (malloc, free, etc.)
- Structs and unions

Ask on Piazza if you are confused or stuck!

How to Develop Software Like a Pro

Topics

- Software Management
  - Source Control
  - Makefiles
- Forging Your Own Foot Armor
  - Avoiding Common C Programming Difficulties
  - Unit Tests
  - Debugging Techniques
- Interactive Exercises From Challenge 0
  - Parser Bug Hunt!
  - Parser Code Review

Source Control

hw3-fixed.c

uh-oh.c

everything.h

hw3-undo-bad-fix.c

hw2_after_office_hours.c

homework1_for_real_this_time-003.c

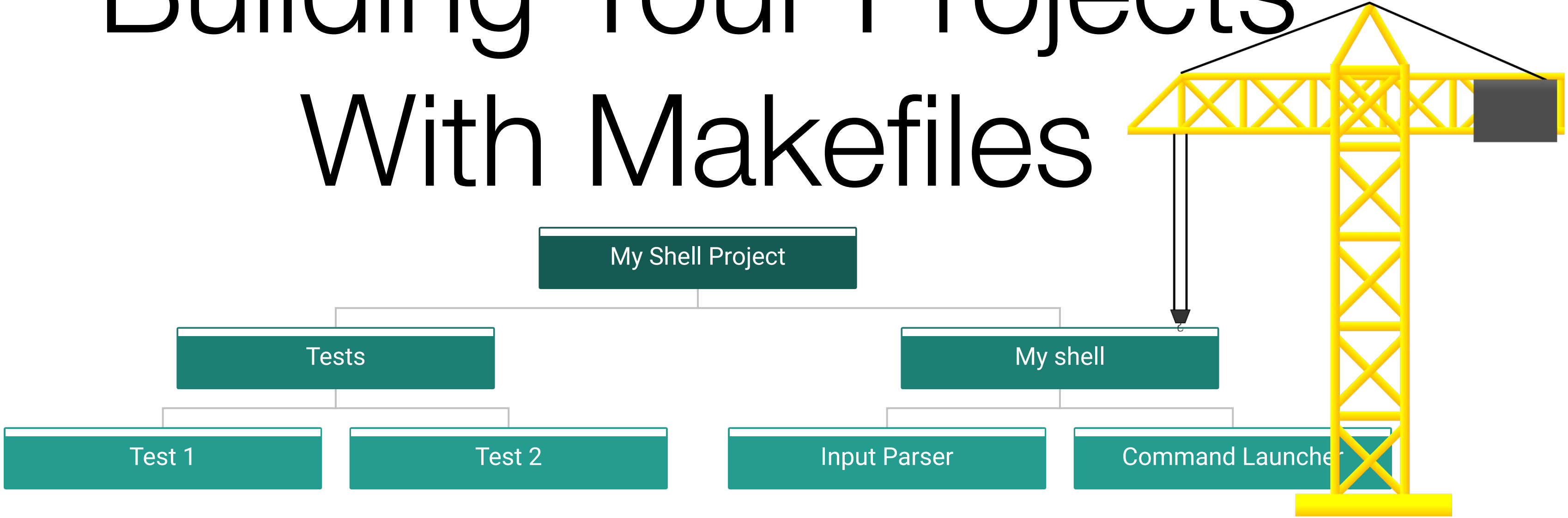
Why Use Source Control

- Code goes through a lot of changes in a project
- Sometimes you need to undo something
- Sometimes you need to redo something from long ago
- Alternatives are confusing to follow:
    - Many renamed versions of one file are hard to track
    - One file with many commented-out old versions is difficult to read

Get Started With Git

- Use any source control you want
- Want a recommendation? Try git
    - Easy to install
    - Decentralized; easy to mirror (e.g., on GitHub)
    - Can submit directly from GitHub to Gradescope
    - It will be easy to get help for this tool on Piazza
- Starting reference: GitHub Hello World Guide
- Ongoing reference: Official Git-SCM Docs
- Use **private repositories** during the semester

Building Your Projects With Makefiles



My Shell Project

Tests

My shell

Test 1

Test 2

Input Parser

Command Launcher

Why Use Makefiles

- `gcc my_code.c -o ./my_code` is easy enough, right?
- It is until it isn't!
  - What if I have multiple files?
  - How do I remember which flags to use?
  - Should I rebuild things that haven't changed?
  - How do I run this project's tests?
- Makefiles automate your build using code!

Makefile Definition

```
# Example from HW0
override CFLAGS := -Wall -Werror -std=gnu99 -O1 -g $(CFLAGS) -I.

# Build the parser .o file
myshell_parser.o: myshell_parser.c myshell_parser.h

# Automatically discover all test files
test_c_files=$(shell find tests -type f -name '*.c' )
test_o_files=$(test_c_files:.c=.o)
test_files=$(test_c_files:.c=)

# The intermediate test .o files shouldn't be auto-deleted in test runs; they
# may be useful for incremental builds while fixing fs.c bugs.
.SECONDARY: $(test_o_files)

.PHONY: clean check checkprogs

# Rules to build each individual test
tests/%: tests/%.o myshell_parser.o
        $(CC) $(LDFLAGS) $+ $(LOADLIBES) $(LDLIBS) -o $@

# Build all of the test programs
checkprogs: $(test_files)

# Run the test programs
check: checkprogs
        tests/run_tests.sh $(test_files)

clean:
        rm -f *.o $(test_files) $(test_o_files)
```

Makefile Definition

```
# Example from HW0
override CFLAGS := -Wall -Werror -std=gnu99 -O1 -g $(CFLAGS) -I.

# Build the parser .o file
myshell_parser.o: myshell_parser.c myshell_parser.h

# Automatically discover all test files
test_c_files=$(shell find tests -type f -name '*.c' )
test_o_files=$(test_c_files:.c=.o)
test_files=$(test_c_files:.c=)

# The intermediate test .o files shouldn't be auto-deleted in test runs;
# may be useful for incremental builds while fixing fs.c bugs.
.SECONDARY: $(test_o_files)

.PHONY: clean check checkprogs

# Rules to build each individual test
tests/%: tests/%.o myshell_parser.o
        $(CC) $(LDFLAGS) $+ $(LOADLIBES) $(LDLIBS) -o $@

# Build all of the test programs
checkprogs: $(test_files)

# Run the test programs
check: checkprogs
        tests/run_tests.sh $(test_files)

clean:
        rm -f *.o $(test_files) $(test_o_files)
```

Implicit Variable Names

CFLAGS holds flags used to *compile* C files.

For C programs, *compilation* and *linking* are separate steps.

LDFLAGS is the name for flags used at *link* time.

LOADLIBES and LDLIBS contain extra libraries use while linking.

CC defines which C compiler to use (e.g., gcc, clang, icc, etc.)

Find more at https://www.gnu.org/software/make/manual/html_node/Implicit-Variables.html

Makefile Definition

```
# Example from HW0
override CFLAGS := -Wall -Werror -std=gnu99 -O1 -g $(CFLAGS) -I.

# Bui
myshe                                              hell_parser.h

# Aut
test_                                              f -name '*.c' )
test_
test_

# The intermediate test .o files shouldn't be auto-deleted in test runs;
# may be useful for incremental builds while fixing fs.c bugs.
.SECONDARY: $(test_o_files)

.PHONY: clean check checkprogs

# Rules to build each individual test
tests/%: tests/%.o myshell_parser.o
	$(CC) $(LDFLAGS) $+ $(LOADLIBES) $(LDLIBS) -o $@

# Build all of the test programs
checkprogs: $(test_files)

# Run the test programs
check: checkprogs
	tests/run_tests.sh $(test_files)

clean:
	rm -f *.o $(test_files) $(test_o_files)
```

To save future-you:
Don’t ignore warnings!

Implicit Variable Names

CFLAGS holds flags used to *compile* C files.

For C programs, *compilation* and *linking* are separate steps.

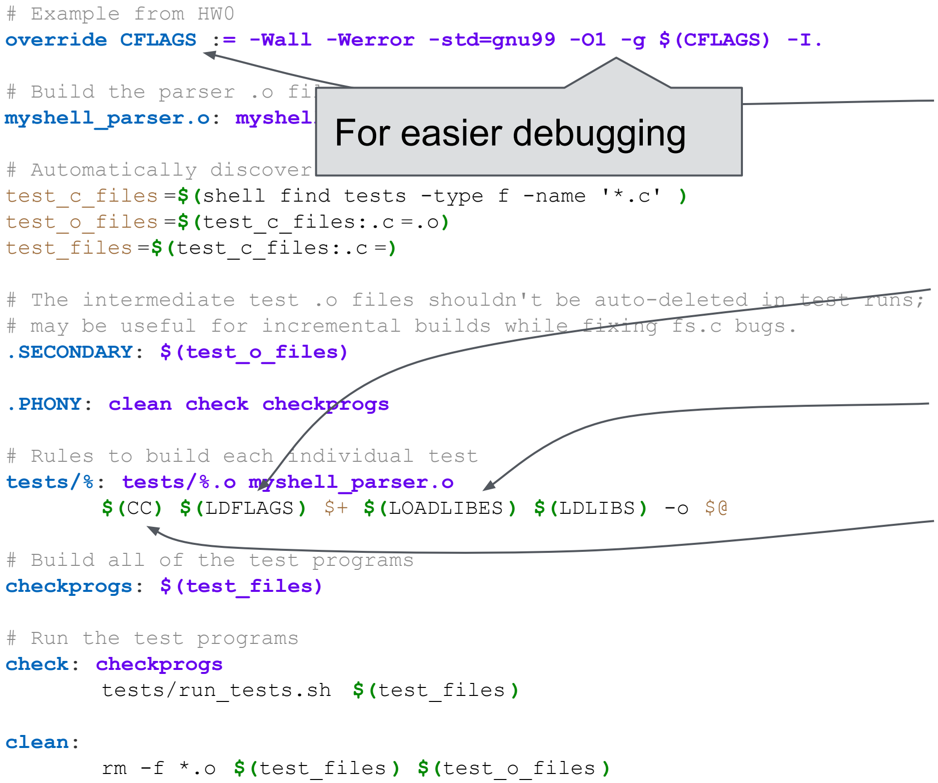
LDFLAGS is the name for flags used at *link* time.

LOADLIBES and LDLIBS contain extra libraries use while linking.

CC defines which C compiler to use (e.g., gcc, clang, icc, etc.)

Find more at https://www.gnu.org/software/make/manual/html_node/Implicit-Variables.html

Makefile Definition



```
# Example from HW0
override CFLAGS := -Wall -Werror -std=gnu99 -O1 -g $(CFLAGS) -I.

# Build the parser .o fi
myshell_parser.o: myshel

# Automatically discover
test_c_files=$(shell find tests -type f -name '*.c' )
test_o_files=$(test_c_files:.c=.o)
test_files=$(test_c_files:.c=)

# The intermediate test .o files shouldn't be auto-deleted in test runs;
# may be useful for incremental builds while fixing fs.c bugs.
.SECONDARY: $(test_o_files)

.PHONY: clean check checkprogs

# Rules to build each individual test
tests/%: tests/%.o myshell_parser.o
	$(CC) $(LDFLAGS) $+ $(LOADLIBES) $(LDLIBS) -o $@

# Build all of the test programs
checkprogs: $(test_files)

# Run the test programs
check: checkprogs
	tests/run_tests.sh $(test_files)

clean:
	rm -f *.o $(test_files) $(test_o_files)
```

For easier debugging

Implicit Variable Names

CFLAGS holds flags used to *compile* C files.

For C programs, *compilation* and *linking* are separate steps.

LDFLAGS is the name for flags used at *link* time.

LOADLIBES and LDLIBS contain extra libraries use while linking.

CC defines which C compiler to use (e.g., gcc, clang, icc, etc.)

Find more at https://www.gnu.org/software/make/manual/html_node/Implicit-Variables.html

Makefile Definition

```
# Example from HW0
override CFLAGS := -Wall -Werror -std=gnu99 -O1 -g $(CFLAGS) -I.

# Build the parser .o file
myshell_parser.o: myshell_parser.c myshell_parser.h

# Automatically discover all test files
test_c_files=$(shell find tests -type f -name '*.c' )
test_o_files=$(test_c_files:.c=.o)
test_files=$(test_c_files:.c=)

# The intermediate test .o files shouldn't be auto-deleted in test runs;
# may be useful for incremental builds while fixing fs.c bugs.
.SECONDARY: $(test_o_files)

.PHONY: clean check checkprogs

# Rules to build each individual test
tests/%: tests/%.o myshell_parser.o
        $(CC) $(LDFLAGS) $+ $(LOADLIBES) $(LDLIBS) -o $@

# Build all of the test programs
checkprogs: $(test_files)

# Run the test programs
check: checkprogs
        tests/run_tests.sh $(test_files)

clean:
        rm -f *.o $(test_files) $(test_o_files)
```

Build Rules

A build target goes to the left of a colon:

- It describes *what will be created*
- It can be run as an argument to make. e.g., *make clean*

A .PHONY target doesn't actually create a file. It only adds the make argument.

Find more at https://www.gnu.org/software/make/manual/html_node/Rules.html

Makefile Definition

```
# Example from HW0
override CFLAGS := -Wall -Werror -std=gnu99 -O1 -g $(CFLAGS) -I.

# Build the parser .o file
myshell_parser.o: myshell_parser.c myshell_parser.h

# Automatically discover all test files
test_c_files=$(shell find tests -type f -name '*.c' )
test_o_files=$(test_c_files:.c=.o)
test_files=$(test_c_files:.c=)

# The intermediate test .o files shouldn't be auto-deleted in test runs;
# may be useful for incremental builds while fixing fs.c bugs.
.SECONDARY: $(test_o_files)

.PHONY: clean check checkprogs

# Rules to build each individual test
tests/%: tests/%.o myshell_parser.o
        $(CC) $(LDFLAGS) $+ $(LOADLIBES) $(LDLIBS) -o $@

# Build all of the test programs
checkprogs: $(test_files)

# Run the test programs
check: checkprogs
        tests/run_tests.sh $(test_files)

clean:
        rm -f *.o $(test_files) $(test_o_files)
```

Build Rules

Build dependencies go to the right of a colon:

- They are what the target depends on.
- All dependencies are updated before building the rule.

Find more at https://www.gnu.org/software/make/manual/html_node/Rules.html

Makefile Definition

```
# Example from HW0
override CFLAGS := -Wall -Werror -std=gnu99 -O1 -g $(CFLAGS) -I.

# Build the parser .o file
myshell_parser.o: myshell_parser.c myshell_parser.h

# Automatically discover all test files
test_c_files=$(shell find tests -type f -name '*.c' )
test_o_files=$(test_c_files:.c=.o)
test_files=$(test_c_files:.c=)

# The intermediate test .o files shouldn't be auto-deleted in test runs;
# may be useful for incremental builds while fixing fs.c bugs.
.SECONDARY: $(test_o_files)

.PHONY: clean check checkprogs

# Rules to build each individual test
tests/%: tests/%.o myshell_parser.o
	$(CC) $(LDFLAGS) $+ $(LOADLIBES) $(LDLIBS) -o $@

# Build all of the test programs
checkprogs: $(test_files)

# Run the test programs
check: checkprogs
	tests/run_tests.sh $(test_files)

clean:
	rm -f *.o $(test_files) $(test_o_files)
```

Build Rules

Build recipes go under the target:

- They are what is executed to build the target.
- There can be more than one command in a recipe
- Recipes MUST be indented by **one tab** (not spaces)

Find more at https://www.gnu.org/software/make/manual/html_node/Rules.html

Makefile Definition

```
# Example from HW0
override CFLAGS := -Wall -Werror -std=gnu99 -O1 -g $(CFLAGS) -I.

# Build the parser .o file
myshell_parser.o: myshell_parser.c myshell_parser.h

# Automatically discover all test files
test_c_files=$(shell find tests -type f -name '*.c' )
test_o_files=$(test_c_files:.c=.o)
test_files=$(test_c_files:.c=)

# The intermediate test .o files shouldn't be auto-deleted in test runs;
# may be useful for incremental builds while fixing fs.c bugs.
.SECONDARY: $(test_o_files)

.PHONY: clean check checkprogs

# Rules to build each individual test
tests/%: tests/%.o myshell_parser.o
        $(CC) $(LDFLAGS) $+ $(LOADLIBES) $(LDLIBS) -o $@

# Build all of the test programs
checkprogs: $(test_files)

# Run the test programs
check: checkprogs
        tests/run_tests.sh $(test_files)

clean:
        rm -f *.o $(test_files) $(test_o_files)
```

Other Notes

Some built-in variables are often used:

- $+ inserts all dependencies of the rule
- $@ inserts the rule’s target

Find more at https://www.gnu.org/software/make/manual/html_node/Automatic-Variables.html

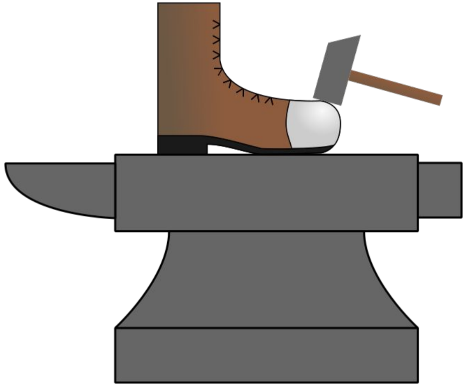
Some implicit rules are often used:

- If no recipe is defined for a target, *make* automatically uses one for that type
- Notice myshell_parser.o has no recipe

Find more at https://www.gnu.org/software/make/manual/html_node/Catalogue-of-Rules.html#Catalogue-of-Rules

Forging Your Own Foot Armor

or: How I Learned to Stop Worrying and Love My Tools



What Can Go Wrong in C

- “It works on my machine but not in the autograder”
- Segmentation faults
  - Or no segmentation fault when you expected one!
- Depend on uninitialized variables
- Data loss through type casts
- Buffer overruns
- Infinite loops
- An if-statement is applied to the wrong scope
- Incomplete switch-case block
- Memory leaks
- …

What Can Go Wrong in C

- “It works on my machine but not in the autograder”
- Segr
  - C cted one!
- Depend on uninitialized variables
- Data loss through type casts
- Buffer overruns
- Infinite loops
- An if-statement is applied to the wrong scope
- Incomplete switch-case block
- Memory leaks
- …

> This was a very common issue in the past.
> We hope that the student environment container will help.
> May still be *caused by* other items in this list (example later).

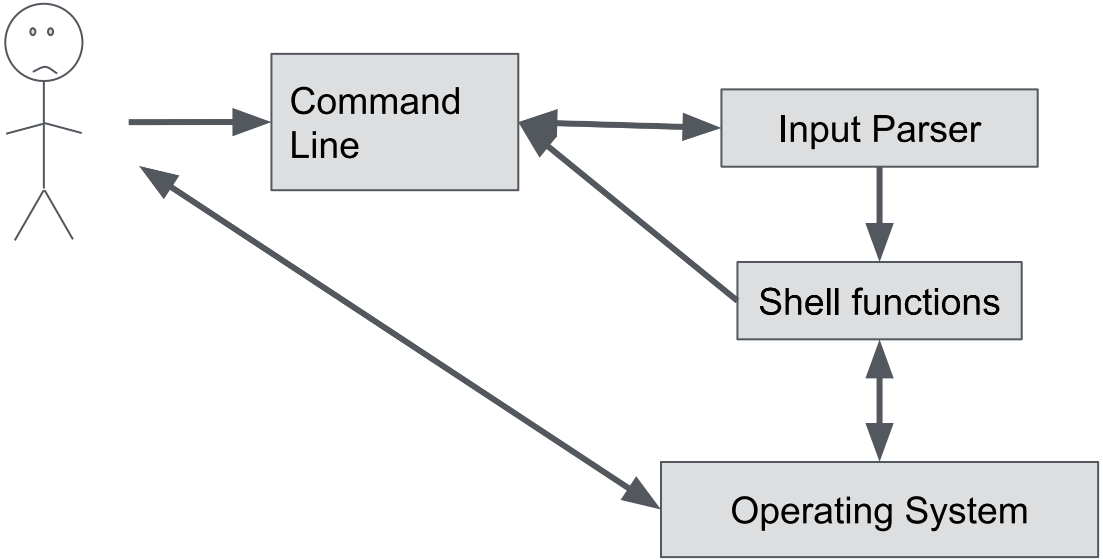
Tools to the Rescue!

- Novice or expert, everyone writes bugs
- Tooling helps find your bugs
- Static Techniques:
    - Compilers emit warnings. Don’t ignore them.
    - Run a static analyzer (e.g. cppcheck, clang-tidy)
- Dynamic Techniques:
    - Use sanitizers (ASAN and UBSAN in your env)
    - Use valgrind (available in your student env)

Helpful Practices

- Did I mention compilers emit warnings?
  - **Don’t ignore them**! Use -Werror to enforce them.
- Develop a test suite as you work.
  - Don’t *replace* tests. *Add* new tests.
  - Run your tests after every change.
- Use a style guide. E.g., Linux Kernel Coding Style
  - Style rules are often written with bugs in mind.
  - Consistent code is easier to review.
- Comment *why* you’re doing something, not *what* you’re doing (the code already says what you’re doing)

Unit Testing

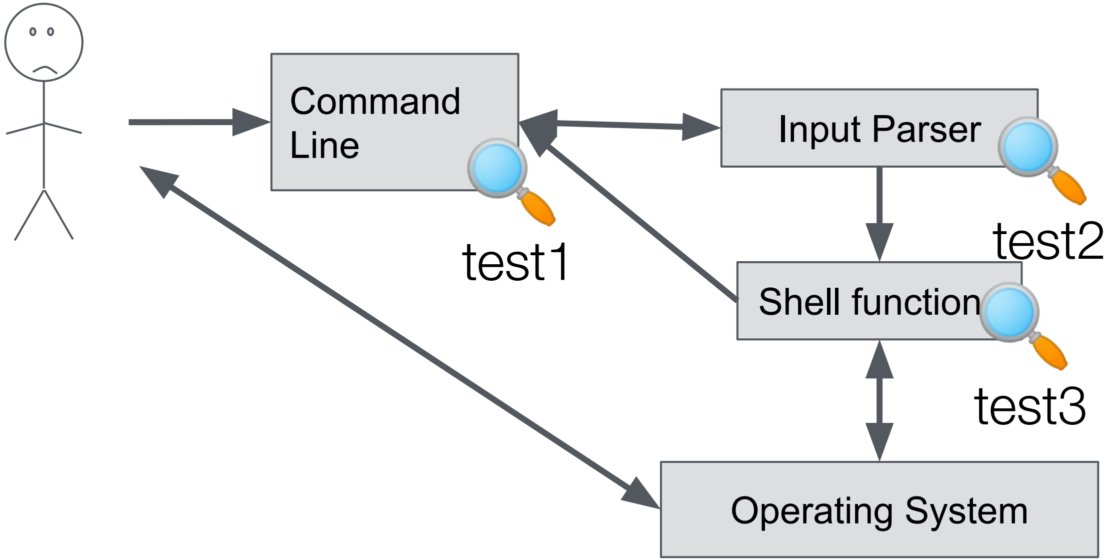


How do you find the bug?

- Watch for side effects in the OS?
- Look for bad output in UI?

This slide is adapted from Prof. Egele's Boston University EC440 lectures

Unit Testing



How do you find the bug?

- Write one or more programs that test specific parts of your code
- Supplements other types of tests (e.g. integration testing)
- This is what we did in hw0!

This slide is adapted from Prof. Egele's Boston University EC440 lectures

Unit Testing - HW0

```c
// Omitted header for brevity. This test is provided with the hw0 prompt
int main(void) {
	struct pipeline* my_pipeline = pipeline_build("ls\n");

	// Test that a pipeline was returned
	TEST_ASSERT(my_pipeline != NULL);
	TEST_ASSERT(!my_pipeline->is_background);
	TEST_ASSERT(my_pipeline->commands != NULL);

	// Test the parsed args
	TEST_ASSERT(strcmp("ls",
	                   my_pipeline->commands->command_args[0]) == 0);
	TEST_ASSERT(my_pipeline->commands->command_args[1] == NULL);

	// Test the redirect state
	TEST_ASSERT(my_pipeline->commands->redirect_in_path == NULL);
	TEST_ASSERT(my_pipeline->commands->redirect_out_path == NULL);

	// Test that there is only one parsed command in the pipeline
	TEST_ASSERT(my_pipeline->commands->next == NULL);

	pipeline_free(my_pipeline);
}
```

A common **testing pattern**:

1. Set up your preconditions (no setup needed here)
2. Interact with your system under test (parser)
3. Assert on your expected postconditions
4. Tear down before the next test

What to assert:

- Unexpected behavior does not occur
- Expected behavior does occur

Unit Testing - HW1

What should I remove from my project to test HW1?

- Nothing! Your finished HW0 solution is the start of HW1!
- Your existing HW0 tests are still valid for HW1.
- Just add a new .c file for each test in the tests/ directory
    - It’s also possible to have multiple tests per file, but for this assignment, one test per file is recommended
- Our `make check` recipe automatically finds all tests

Interactive Excercises

Bug Hunt!

- Let’s look at a test for a challenge 0 solution
- The test passes initially
- ...But it fails with a small modification
- How can we find the source of the failure?

Bug Hunt!

Let’s Try:

- Debugger
- Analysis tools

Code Review

Review each other's code for challenge 0.

- Challenge 0 implementation:
    - Do not copy this code from each other.
    - Point out potential bugs
    - Note readability concerns to each other
- Challenge 0 tests:
    - What is missing?
    - See any great tests? *THOSE are okay to share.*
- See something interesting or confusing? Post and discuss on Piazza

Derived Resources Used In These Slides

- Many of these slides are derived from slides provided by Prof. Egele at Boston University
- Git Logo by Jason Long is licensed under the Creative Commons Attribution 3.0 Unported License.