Lecture 02
Unix Shell & C Compilation

CS211 – Fundamentals of Computer Programming II

Branden Ghena – Spring 2023

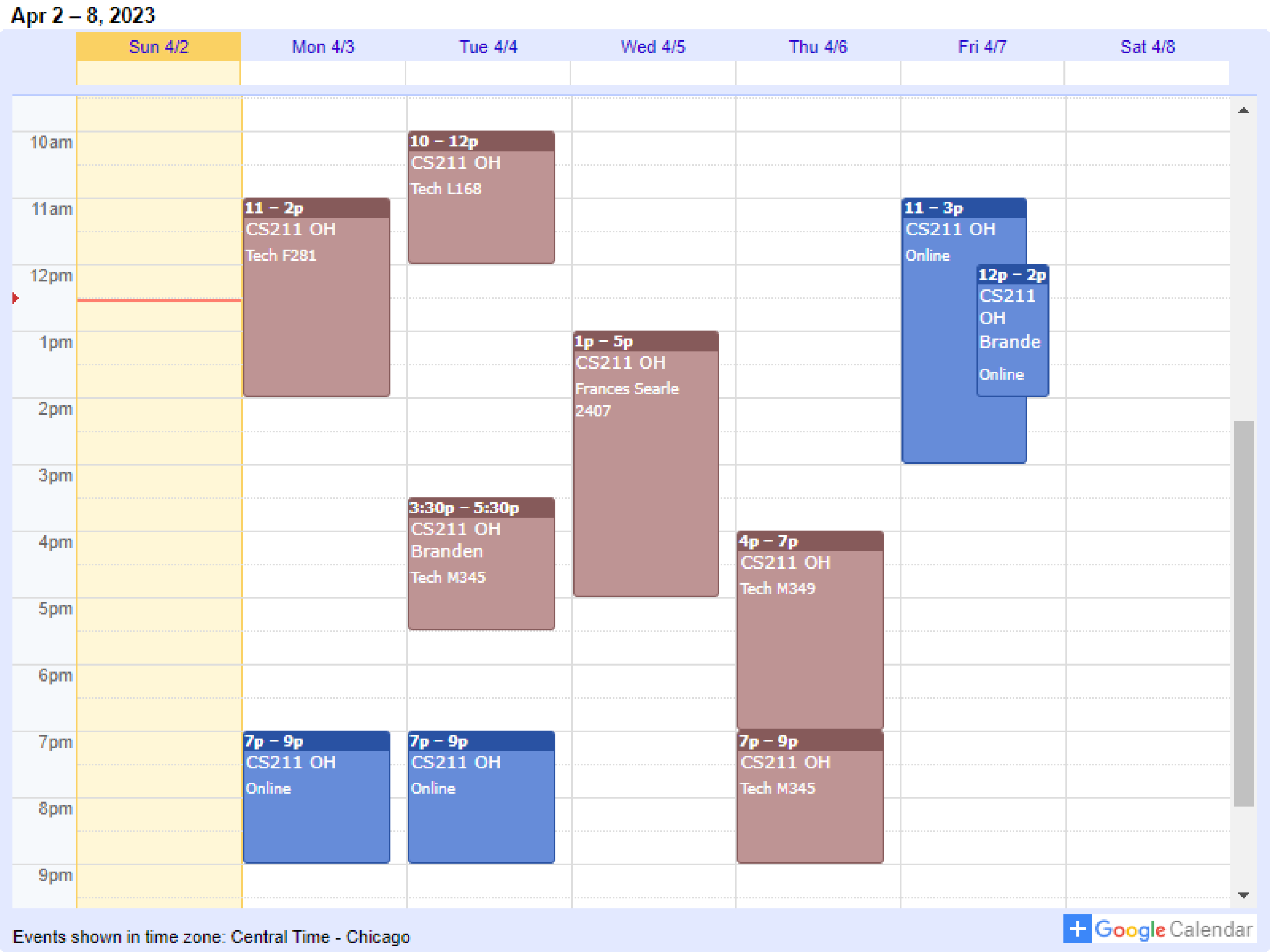
Slides adapted from:
Jesse Tov

Northwestern

Administrivia

- Office hours have started!
  - Check Canvas homepage for calendar
  - I have office hours right after class today!

- Everyone should have Piazza access
  - Email me ASAP if you don't



Assignments

- EX1 due today (88%+ of you are done)
  - Need to buy the textbook unfortunately
  - Remember: no late submission for exercises

- EX2 due Thursday (32%+ completed)
  - A little deeper into C programming: Branches and Loops

- Lab1 due Thursday (31%+ completed)
  - SSH access to lab servers for C programming
  - Using Linux command line
  - Submitted to Gradescope

Today's Goals

- Introduction to working in Unix shell (command line)
- Understand the C compilation process
- Continue exploring C programming
  - Iteration
  - Input and Output

Getting the examples from lecture

- First, make your own cs211 directory to store class stuff in
  - `cd ~/`
  - `mkdir cs211`

- The files for this class are in a zipped tarball (just like a zip file)
  - We can extract them right into your cs211/ directory

  - `cd ~/cs211/`
  - `tar -xvkf ~cs211/lec/02_shell_compilation.tgz`
  - `cd 02_shell_compilation`

  - What does that command do?: https://explainshell.com/explain?cmd=tar+-xvkf+%7Ecs211%2Flec%2F02_shell_compilation.tgz

Outline

- **Unix Shell**
  - **Navigation**
  - Working with files

- Compilation
  - Separate Compilation
  - Makefiles
  - Pre-processor

- More C syntax
  - Computing Fibonacci Numbers
  - Iteration
  - Input and Output
  - Other C Syntax

How do you get a Unix shell?

- Have a MacOS or Linux computer
  - Or set up Windows Subsystem for Linux (WSL) on Windows

- Install Virtualbox and Linux
  - Installing Ubuntu is free and only takes twenty minutes

- Log in to a class server remotely!
  - This is what we'll do for CS211
  - Lab01 teaches you how to do this

Command line interfaces

- Text-based commands

- Positives
  - It’s easy to be precisely clear about what you want and how things are configured

- Negatives
  - How do you remember everything?

- Reality
  - There will be a few dozen commands you’ll memorize (after practice)
  - And you’ll learn how to look up everything else

Commands for moving between directories

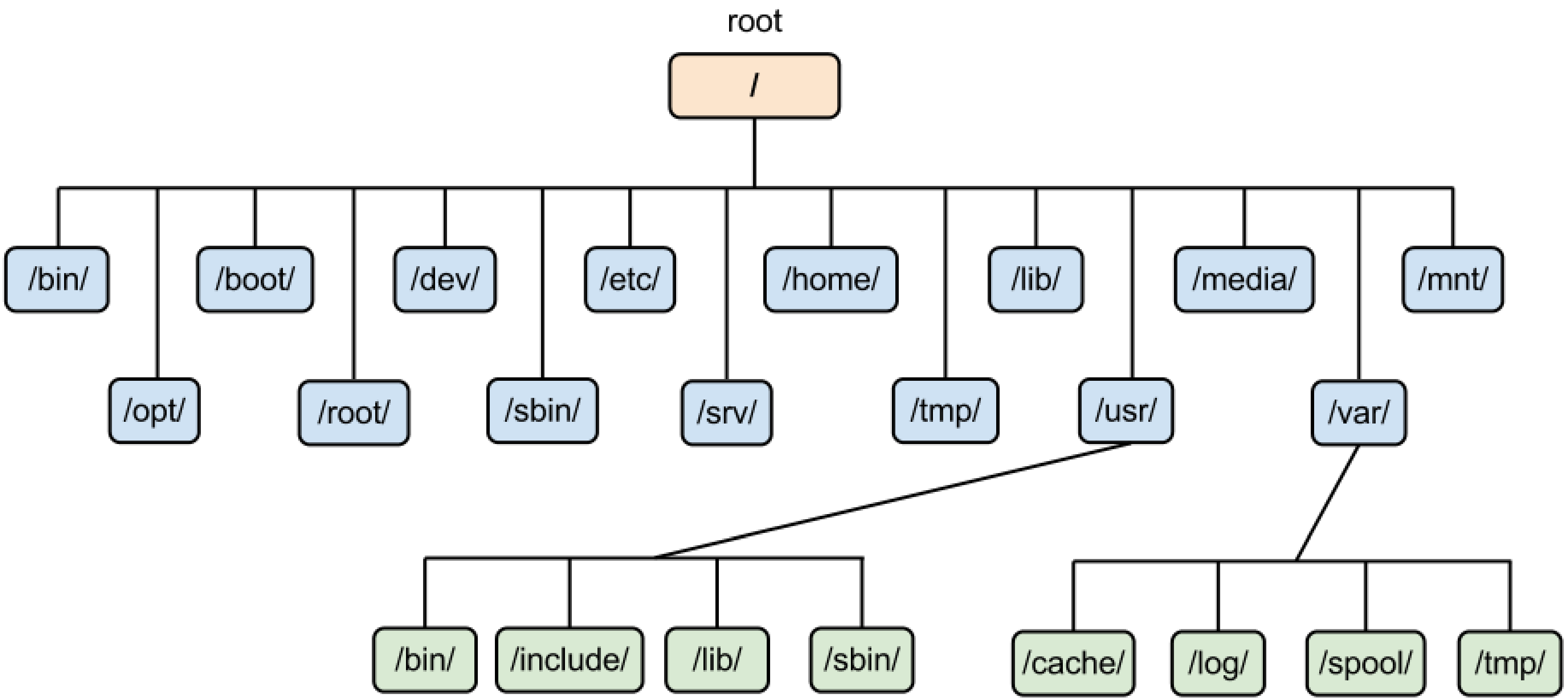
- Directory structure and moving through it
  - `ls`
    - Lists files in the current directory
  - `cd`
    - Change directory
  - `pwd`
    - Prints the path of the current directory

- Mis-typing something
  - "Command not found" means you tried to run something invalid
  - `fish: somecommandyoumistyped: command not found...`

Live command-line demo!!!

- To do:
  - Log in
  - Move around with commands
  - Fail at some command
  - Tab completion
  - Get files from lecture!

Directory structure in Linux



- Example: `/usr/bin/` is the path to user-installed programs

Special paths

`.`	the current directory
`..`	the parent of the current directory
`../../`	the parent of the parent of the current directory
`../../../`	and so on…
`-`	the previous directory you were in before the current one
`~/`	the home directory of the current user (your home)
`~cs211`	the home directory of the user cs211 (works for any user, but you'll probably won't interact with other users)
`/`	the root directory (analogous to `C:\` on windows)

Relative vs absolute paths

- Relative paths are relative to the current directory
  - `../`
  - `src/`
  - `../../code/src/../build/`

- Absolute paths have the full path name to the location
  - `/home/branden/`
  - `/home/branden/cs213/code/`
  - `/home/branden/cs213/code/src/../build/`

Wildcard in path names

- Sometimes you're not sure exactly what the name is
  - Or there might be multiple files that you want to interact with simultaneously

- The wildcard symbol, *, replaces any number of characters in a path name

- Examples
  - `ls /home/*/` List all files in all user's home directories
  - `ls ~/cs21*/` List all files in any directory starting with cs21
  - `ls code/src/*.c` List all files that end with ".c" in code/src/

Tab Completion

- Typing takes tooooooooo looooooonnnnggg
  - Solution, let the computer guess what you're trying to type

- Pressing tab while part-way through typing just about anything in terminal will tab-complete it for you
  - As long as you have typed enough characters so that only one option remains, it will complete it
  - If multiple options remain, it will stop trying

- Also, up-arrow gets you the previously typed command
  - And you can edit that, if that's faster

Outline

- **Unix Shell**
  - Navigation
  - **Working with files**
- Compilation
  - Separate Compilation
  - Makefiles
  - Pre-processor
- More C syntax
  - Computing Fibonacci Numbers
  - Iteration
  - Input and Output
  - Other C Syntax

Working with files

- `cat path`
  - Prints out the contents of the file
- `mv path1 path2`
  - Moves a file from path1 to path2
- `cp path1 path2`
  - Copies a file from path1 to path2
- `rm path`
  - Deletes (removes) a file

Editing files

- There are many different terminal text editors
  - And there are holy wars about why one is *best*
  - **There is no best. Just use whatever you like**

- Example editors
  - Vim, Emacs, Nano

- In CS211, I’ll be teaching you using the Micro text editor
  - Occasionally I’ll open vim by accident. Someone yell at me when I do
  - https://micro-editor.github.io/

Editing with Micro

- micro filename
  - Opens micro, editing filename

- Works just like any text editor you’ve used
  - Mouse moves the cursor around, as do the arrow keys
  - Typing makes text appear
    - (This isn’t true in some shell editors, looking at you vim)

  - Ctrl-s save the file
  - Ctrl-o open a file
  - Ctrl-q quit

Live command-line demo 2!!

- To do:
  - Make directories
  - Edit a file
  - Move a file
  - Use a command with flags

Cancelling a command

- Ctrl-C stops *most* things from running
  - Ctrl key and C key both at once

- If you have C code that's stuck in an infinite loop, Ctrl-C will stop it

- Note: this means Ctrl-C isn't usually copy
  - Except it does work as copy in Micro!
    (but that means it won't stop Micro from running)

Command flags

- `man`
  - Opens the manual pages for a program
  - Example: `man ls`

- Flags are configurations for a command that change what it does
  - `ls –l` lists files in the current directory in a vertical list with details
  - `ls –t` sorts the ls output by most recently modified
  - `ls –l –t` does both

- You can type multiple flags after a single dash
  - `ls –lt` is equivalent to `ls –l –t` is equivalent to `ls -tl`

Searching for things

- `grep –r "text" *`
  - Explanation
    - Grep prints lines matching a pattern
    - The pattern in this case is "text"
    - `–r` means search recursively, i.e. in this directory and all subdirectories
    - * means to search in any file in the current directory
  - Summary
    - Search all the files here and below for the word "text"

Don’t be overwhelmed!!!!

- You have plenty of time to learn this
- Lab01 guides you through the same kinds of commands I did today, step by step
- Practice is the only thing that will really help
  - And CS211 will give you plenty of practice

Helpful guides

- Great lecture notes on using the shell
  - https://swcarpentry.github.io/shell-novice/

- Tool to explain various shell command syntax
  - https://explainshell.com/

- Tool to explain how to use various shell commands
  - Just type the command into the box at the top
  - https://tldr.ostera.io/

Shell command: sudo

- Superuser do
  - Executes a command with special administrator privilege (superuser)
  - Necessary for installing new programs and modifying the OS

- Run it before a command to execute that command as a superuser
  - Example: `sudo rm -rf /` (don't run this!)

- You can only use `sudo` on computers where you are an admin
  - Only use with caution and care. It can destroy your computer

  - You'll never need it for class stuff
  - You are NOT an admin on the class servers! (neither am I)
  - You might see it in stack overflow answers (won't solve 211 problems though)

sudo example

```
branden@moore:~% sudo echo "Sorry Pred, I'm testing this for CS211."

We trust you have received the usual lecture from the local System
Administrator. It usually boils down to these three things:

    #1) Respect the privacy of others.
    #2) Think before you type.
    #3) With great power comes great responsibility.

[sudo] password for branden:
```

sudo example

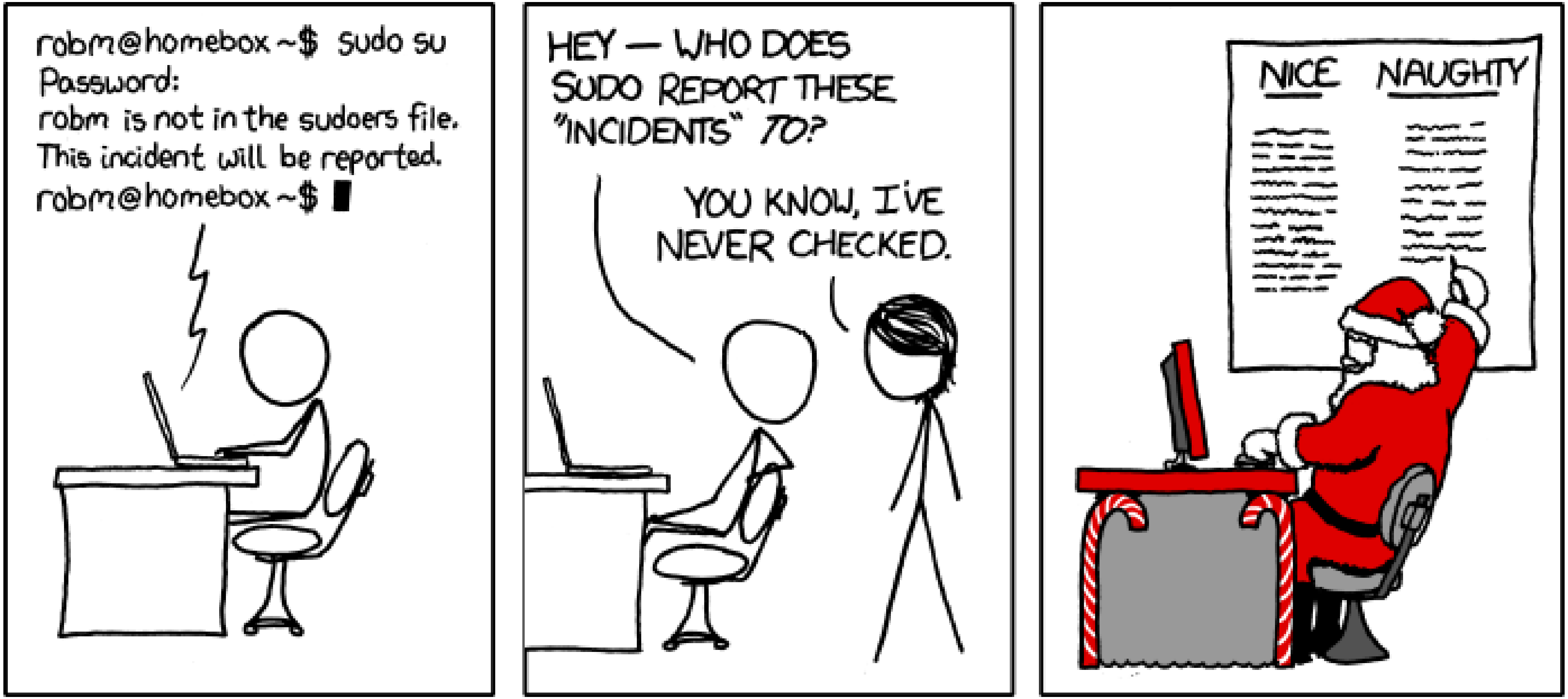
```
branden@moore:~% sudo echo "Sorry Pred, I'm testing this for CS211."

We trust you have received the usual lecture from the local System
Administrator. It usually boils down to these three things:

    #1) Respect the privacy of others.
    #2) Think before you type.
    #3) With great power comes great responsibility.

[sudo] password for branden:
branden is not in the sudoers file.  This incident will be reported.
branden@moore:~ [1]%
```

Break + relevant xkcd



https://xkcd.com/838/

Outline

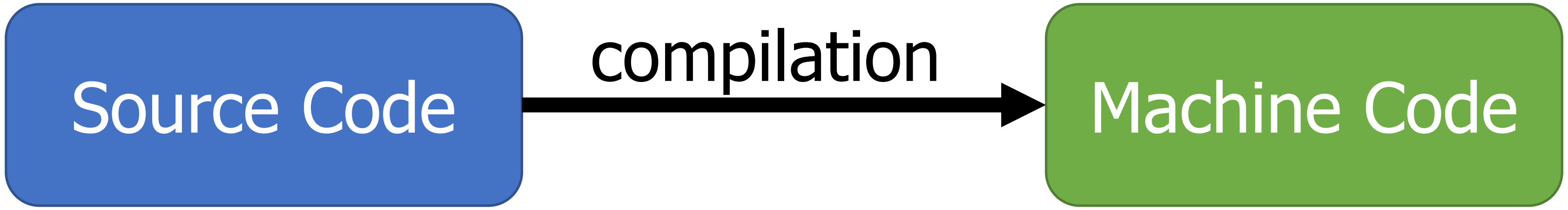
- Unix Shell
  - Navigation
  - Working with files

- **Compilation**
  - Separate Compilation
  - Makefiles
  - Pre-processor

- More C syntax
  - Computing Fibonacci Numbers
  - Iteration
  - Input and Output
  - Other C Syntax

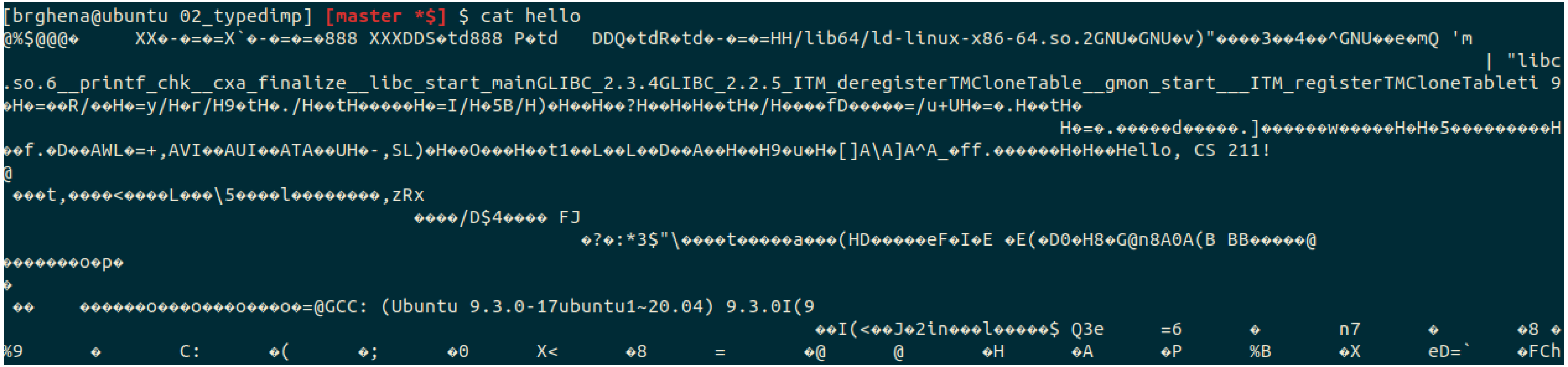
How do you "run" C code?

- First, the C code needs to be translated
  - From human-readable source code
  - To machine code capable of being executed on a particular machine (definitely not human readable)
- This translation process is called "compiling"
  - The tool that does it is a "compiler"



What does machine code look like?

- Just a bunch of numbers
  - Your text editor would interpret those numbers as random characters



- The computer processor reads the numbers to figure out which instruction to run
  - This is a version of assembly code
  - See CS213 for *way* more details

Compiling a C program

- The compiler we'll use is referred to as `cc`
  - Short for C Compiler
  - It takes in C source code and outputs *executable* machine code

- `cc hello.c`
- ```
  ls
  a.out hello.c
  ```
- ```
  ./a.out
  Hello, CS 211!
  ```

Don't memorize this. You won't be running cc manually.

Compiling a C program

- a.out is the default name, but we probably want to use something more memorable
- The `-o` flag specifies the output filename for the compiler

- `cc -o hello hello.c`
- `ls`
  `hello hello.c`

- `./hello`
  `Hello, CS 211!`

Don't memorize this. You won't be running cc manually.

Remember to compile!

- You need to re-compile code every time the source code changes

- You WILL forget to do this at some point
  - And you'll run the program but it'll do the old behavior rather than the new things you've written

- Compile often!
  - Keep multiple windows open to make this easier
  - I write a handful of lines of C code, then compile again
    - Way easier to find one or two mistakes now than deal with MANY later

IMPORTANT: compile often!

- Important enough that I’ll repeat it

- Keep multiple terminals open
  - One for editing and one for compiling

- Compile every few lines of C code you write
  - Maybe every time you finish a function

- Compilation points out errors in your code for you!
  - But it can get overwhelming if you don’t run it until the end

Outline

- Unix Shell
  - Navigation
  - Working with files
- **Compilation**
  - **Separate Compilation**
  - Makefiles
  - Pre-processor
- More C syntax
  - Computing Fibonacci Numbers
  - Iteration
  - Input and Output
  - Other C Syntax

Real-world projects have multiple files

- You can write code in any number of different C files
  - And combine them together while compiling

- But we need some way to tell C code in one file about the existence of C code in another file
  - Solution: header files (.h)

  - Header files list all the publicly available functions and variables from a C file
    - Usually, there is a .c and .h file for various libraries

  - Header files are `#include`-ed at the top of your C file

Compiling multiple C files

- Each C file is compiled separately
- Then combine multiple together into a single program

- Compilers have a middle step: object files (.o)
  - Still not human readable
  - Meant to be joined together into a single executable

- Object files don't have to be recompiled if their source file hasn't changed
  - This speeds up compilation for large projects!

General C project layout

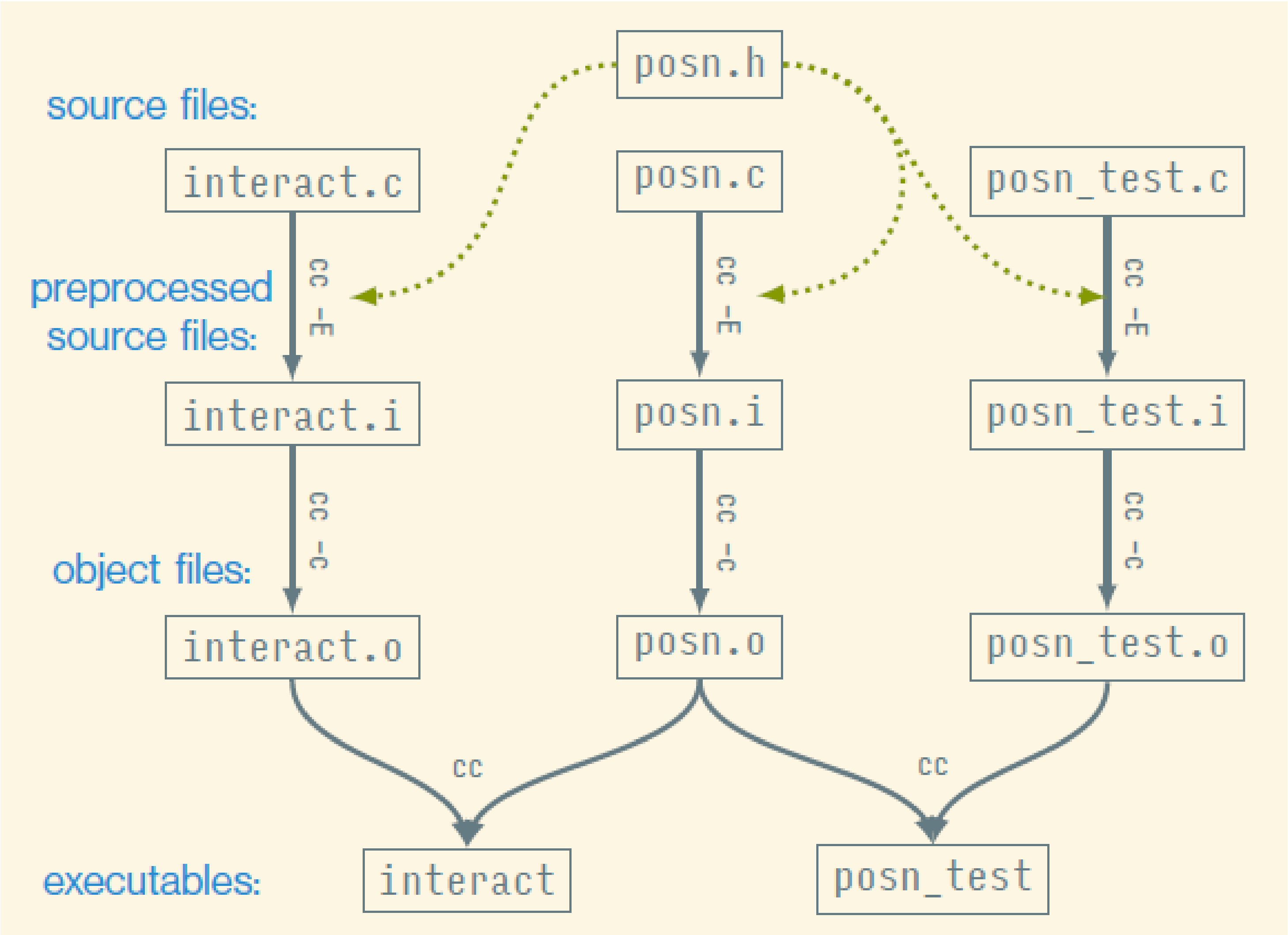
- src/
  - Various code that actually runs your project
- test/
  - Various code that tests your files in src/

- We separate code in src/ into two categories
  - The executable, which has a main() function and not much else
    - Named whatever your executable is, but with a .c
    - Example: interact.c

  - Libraries which have both .c and .h files
    - Example: posn.c and posn.h

Example of multiple compilation

example_project/



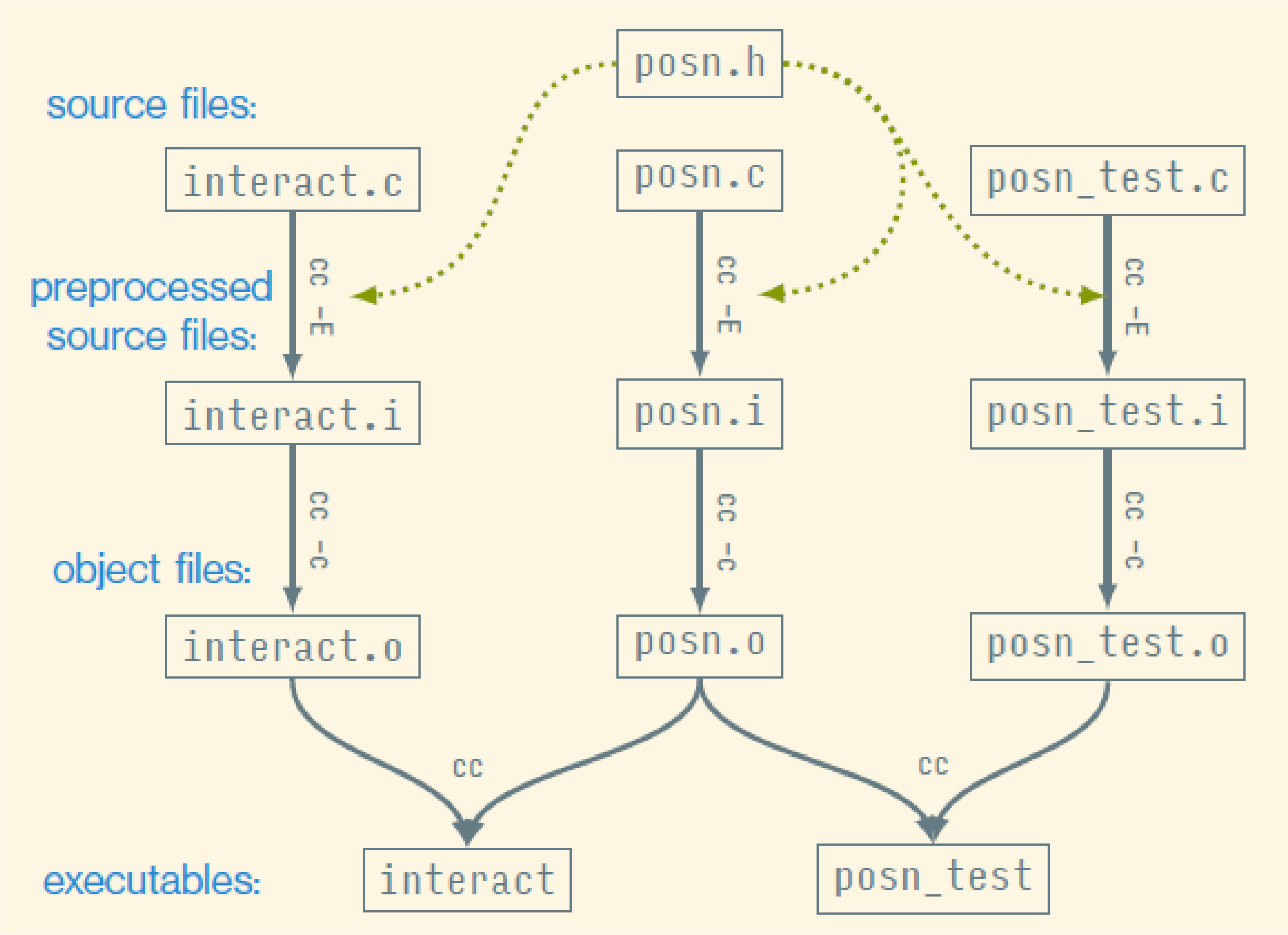
Outline

- Unix Shell
  - Navigation
  - Working with files

- **Compilation**
  - Separate Compilation
  - **Makefiles**
  - Pre-processor

- More C syntax
  - Computing Fibonacci Numbers
  - Iteration
  - Input and Output
  - Other C Syntax

New problem, how do you remember all these steps?



And this doesn't even include various flags we give to the compiler, such as the location of the 211.h library

Simplifying multiple compilation with Make

- Make is a tool for building programs out of multiple source files
  - Allows you to specify goals and requirements as “rules”
  - And then runs the compiler to fulfill those

- To build a file named ⟨`goal`⟩ using make, you run:

  `make ⟨goal⟩`

- `Make` looks around the current directory for a file named Makefile which specifies the various rules
  - We’ll provide the Makefile for you in this class
  - But you’ll have to use `make` to compile your programs

What does a `make` rule look like?

- A rule has a goal and pre-requisites for the goal
  - And then specifies commands to create the goal given the pre-requisites

```
⟨goal⟩: ⟨prereqs⟩. . .
   ⟨commands⟩
   ...
```

- Example:

```
hello: hello.c
   cc -o hello hello.c
```

Always use Make, rather than calling the compiler yourself

- Make is our tool for compiling programs
  - It has rules for how to build the programs using the compiler

- You *could* compile your programs manually
  - But you would need to know the proper flags for the compiler to do so
  - Some programs rely on class-specific libraries for testing and memory management

  - This is a big pain, so just you `make` instead
    - And if you’re curious, you can look at the Makefile to see what the flags we’re providing are

Outline

- Unix Shell
    - Navigation
    - Working with files
- **Compilation**
    - Separate Compilation
    - Makefiles
    - **Pre-processor**
- More C syntax
    - Computing Fibonacci Numbers
    - Iteration
    - Input and Output
    - Other C Syntax

C pre-processor

- Reads in the text of your source code

- Does some initial text-based manipulations to the code
  - Prepares everything for the compiler

C reads files from the top down

- First important thing to know about the pre-processor/compiler
  - They read from the top of the file down
  - Functions that don’t exist when you try to call them are an error

- How would we write this code then?

```
void a(void) {
  b();
}

void b(void) {
  a();
}
```

Function declaration

- You can inform the compiler about functions that will later be defined
  - You are telling the C compiler: “here’s what this other function looks like, you’ll get details about how it works later”
  - Very useful for libraries that you are using

- A function **declaration** in C includes the return type, name, and argument types
  - Examples:

```
void a(int, float);
struct posn read_posn(void);
```

- A function **definition** in C also includes the body of the function

Header files are collections of declarations

- You could manually type out the declaration for each function you want to use at the top of your C file

- Instead, we create "Header files" (.h) that hold all the function declarations for functions in the associated .c file

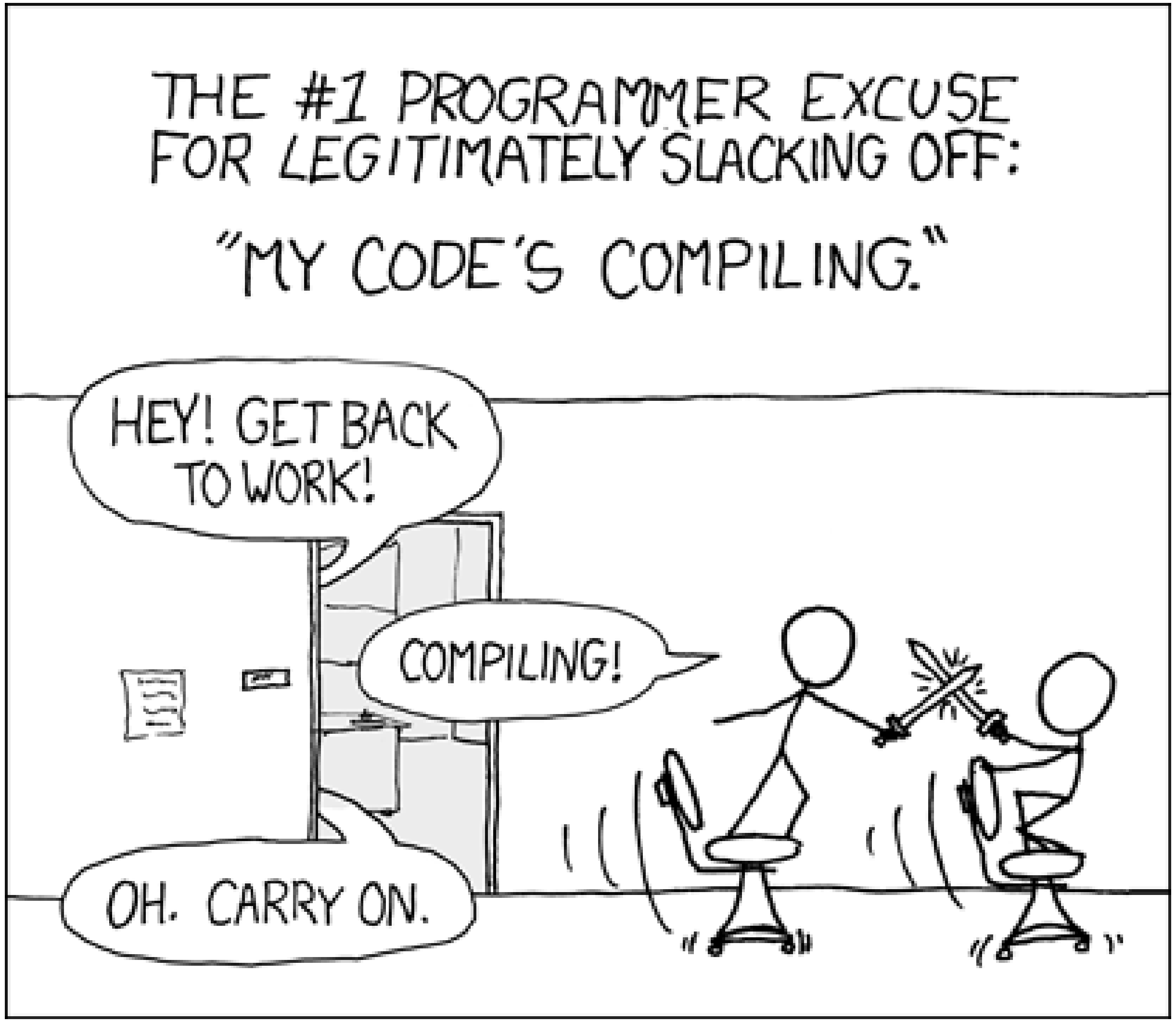
- `#include`-ing a header file tells the pre-processor to paste its contents
  - The same as if you had typed them in the top of the file yourself
  - Leads to weird errors sometimes if you mess up a header file
  - Be sure to only include header files!

Examples

example_project/
preprocessor_example/

- The –E flag tells the compiler to only run the pre-processor
- In example_project/
  - cc –E src/interact.c –o interact.i
    - Note that header files are included
    - Note that some functions are only definitions right now
- Simpler example can be found in preprocessor_example/
  - Run `make` to create client.i and library.i

Break + relevant xkcd



https://xkcd.com/303/

Outline

- Unix Shell
  - Navigation
  - Working with files

- Compilation
  - Separate Compilation
  - Makefiles
  - Pre-processor

- **More C syntax**
  - **Computing Fibonacci Numbers**
  - Iteration
  - Input and Output
  - Other C Syntax

Definition of Fibonacci Function

- $fib(n) = \begin{cases} n, & if\ n < 2; \\ fib(n-2) + fib(n-1), & otherwise \end{cases}$

n	**fib(n)**
0	0
1	1
2	1
3	2
4	3
5	5
6	8
7	13
8	21

Implementing Fibonacci in C

fib.c

```
long fib(int n){
  if (n < 2) {
    return n;
  } else {
    return fib(n - 2) + fib(n - 1);
  }
}
```

$$fib(n) = \begin{cases} n, & \text{if } n < 2; \\ fib(n-2) + fib(n-1), & \text{otherwise} \end{cases}$$

Outline

- Unix Shell
  - Navigation
  - Working with files
- Compilation
  - Separate Compilation
  - Makefiles
  - Pre-processor
- **More C syntax**
  - Computing Fibonacci Numbers
  - **Iteration**
  - Input and Output
  - Other C Syntax

Statements and Conditions aren't enough

- Not all problems are easily solved with recursion

- C, like many programming languages, also has loops
  - Repeats the statements inside it until some condition is met

Iteration with the While Statement

- Syntax

```
while (⟨test-expression⟩) {
  ⟨body-statements⟩
}
```

- Semantics
  1. Evaluate `⟨test-expression⟩` to a `bool`
  2. If the `bool` is *false* then skip to the statement after the `while` loop
  3. Execute `⟨body-statements⟩` (if the `bool` was true)
  4. Go back to step 1

Implementing Fibonacci in C

fib.c

```
long fib_iterative(int n){
    long curr = 0;
    long next = 1;

    while (n > 0) {
        long prev = curr;

        curr = next;
        next = prev + curr;
        n = n - 1;
    }

    return curr;
}
```

$$fib(n) = \begin{cases} n, & if\ n < 2; \\ fib(n-2) + fib(n-1), & otherwise \end{cases}$$

For loops

- For loops allow you to combine iteration and incrementing
  - When you write a for statement like this:

```
for (⟨start-decl⟩; ⟨test-expr⟩; ⟨step-expr⟩) {
  ⟨body-stms⟩
}
```

  - It's as if you'd written this while statement:

```
{
  ⟨start-decl⟩;
  while (⟨test-expr⟩) {
    ⟨body-stms⟩
    ⟨step-expr⟩;
  }
}
```

Modify fib to use a for loop

fib.c

```
long fib_iterative(int n){
    long curr = 0;
    long next = 1;

    while (n > 0) {
        long prev = curr;

        curr = next;
        next = prev + curr;
        n = n - 1;
    }

    return curr;
}
```

$$fib(n) = \begin{cases} n, & if\ n < 2; \\ fib(n-2) + fib(n-1), & otherwise \end{cases}$$

Modify fib to use a for loop

fib.c

```
long fib_iterative(int n){
    long curr = 0;
    long next = 1;
    int i = 0;

    while (i < n) {
        long prev = curr;

        curr = next;
        next = prev + curr;
        i = i + 1;
    }

    return curr;
}
```

$$fib(n) = \begin{cases} n, & if\ n < 2; \\ fib(n-2) + fib(n-1), & otherwise \end{cases}$$

Complete: modify fib to use a for loop

fib.c

```
long fib_iterative(int n){
    long curr = 0;
    long next = 1;

    for (int i = 0; i < n; i = i + 1) {
        long prev = curr;

        curr = next;
        next = prev + curr;
    }

    return curr;
}
```

$$fib(n) = \begin{cases} n, & \text{if } n < 2; \\ fib(n-2) + fib(n-1), & \text{otherwise} \end{cases}$$

Break + Question

- What value will this code return when called as:
  - loop_function(6)
  - loop_function(5)
  - loop_function(3)

```c
int loop_function(int test) {
  int retval = 0;
  while (test < 5) {
    retval = retval + 1;
    test = test + 1;
  }
  return retval;
}
```

Break + Question

- What value will this code return when called as:
  - loop_function(6) **returns 0**
  - loop_function(5)
  - loop_function(3)

```
int loop_function(int test) {
  int retval = 0;
  while (test < 5) {
    retval = retval + 1;
    test = test + 1;
  }
  return retval;
}
```

Break + Question

- What value will this code return when called as:
  - loop_function(6) **returns 0**
  - loop_function(5) **returns 0**
  - loop_function(3)

```
int loop_function(int test) {
  int retval = 0;
  while (test < 5) {
    retval = retval + 1;
    test = test + 1;
  }
  return retval;
}
```

Break + Question

- What value will this code return when called as:
  - loop_function(6) **returns 0**
  - loop_function(5) **returns 0**
  - loop_function(3) **returns 2**

```
int loop_function(int test) {
  int retval = 0;
  while (test < 5) {
    retval = retval + 1;
    test = test + 1;
  }
  return retval;
}
```

Outline

- Unix Shell
  - Navigation
  - Working with files
- Compilation
  - Separate Compilation
  - Makefiles
  - Pre-processor
- **More C syntax**
  - Computing Fibonacci Numbers
  - Iteration
  - **Input and Output**
  - Other C Syntax

printf() function

- The usual way to print in C is the `printf()` function
  - Takes a *format string* followed by arguments to *interpolate* in place of the string's format specifiers

```
printf("(%d, %d)\n", x, y);
```

`%d` format specifier means the argument is an `int`

Prints "`(`" + the value of x + "`,  `" + the value of y + "`)\n`"

- `printf()` is in the stdio.h library, which needs to be `#include`-ed

Example: formatted output

output.c

```c
#include <stdio.h>
int main(void){
  int x = 5;
  double f = 5.1;
  printf("sizeof x: %zu bytes\n", sizeof(x));
  printf("sizeof f: %zu bytes\n", sizeof(f));
  printf("x: %d\nf: %.60e\n", x, f);
}
```

- A format specifier gives the argument's type and maybe some options
  - `%zu` type: `size_t` (the return result of `sizeof`)
  - `%d` type: `int`
  - `%.60e` type: `double`, include 60 digits of precision

How do you learn format specifiers?

- You look them up in a guide!
  - Even I don’t have them memorized…

- `man 3 printf`
  - Runs in the terminal
  - Shows details about printf

- google “printf format specifiers” (this is what I do)
  - cplusplus.com is a good resource
  - https://www.cplusplus.com/reference/cstdio/printf/

Reading user input

- To input numbers in C, use the `scanf()` function

- scanf reads keyboard input, converts it to the require type, and stores it in an existing variable:

```
int x = 0;
scanf("%d", &x);
```

  - Like `printf()`, `scanf()` uses a format string to determine what type to convert the input into
  - `&x` means to pass x's location, not its value (more on this next week)
  - Careful: `scanf()` directives aren't exactly the same as `printf()`

Example: reading input

input.c

```
#include <stdio.h>

double sqr_dbl(double n){
  return n * n;
}

int main(void){
  double d = 0.0;
  scanf("%lf", &d);
  printf("%lf squared is %lf\n", d, sqr_dbl(d));
}
```

Example: reading multiple items

multi_input.c

```
#include <stdio.h>
int main(void){
  int x;
  int y;
  printf("Enter two integers: ");
  scanf("%d%d", &x, &y);
  printf("%d * %d = %d\n", x, y, x * y);
}
```

What if scanf() has an error?

check_input.c

- `scanf()` returns the number of successful conversions

```
#include <stdio.h>
int main(void){
   int x
   int y;
   printf("Enter two integers: ");
   if (scanf("%d%d", &x, &y) != 2) {
     printf("Input error\n");
     return 1;
   }
   printf("%d * %d == %d\n", x, y, x * y);
}
```

Outline

- Unix Shell
  - Navigation
  - Working with files

- Compilation
  - Separate Compilation
  - Makefiles
  - Pre-processor

- **More C syntax**
  - Computing Fibonacci Numbers
  - Iteration
  - Input and Output
  - **Other C Syntax**

C comments

- `//` means a single-line comment
- `/*` starts a multiline comment, which continues until `*/`

- How to use comments effectively
  - Comment "blocks" of code with their purpose
    - Every line is too much
    - Often helpful to write the comments before the code as planning
  - Comment tricky bits of code so you know what it means
    - You + several weeks = "what does that code mean?!"

Logical operators

- || &&
  - Logical OR, and Logical AND
  - `a < 5 && b > 12`
- !
  - Logical NOT
  - `!(a < 5)` equivalent to `(a >= 5)`
- ==
  - Equality test
  - `5 == 5` -> TRUE
  - `16 == -3` -> FALSE
  - Don’t mix it up with assignment (single equals sign)
    - Really common new C programmer mistake

Other operators you'll see around

- `+=   *=   -=   /=`
  - Perform the action of `VAR = VAR operator ARG`
  - `a += 5` -> `a = a + 5`
  - `a *= b` -> `a = a * b`

- `%`
  - Modulus operator
  - Returns the remainder of division
  - `12 % 10` -> 2

- `~ | & ^`
  - Bitwise NOT, OR, AND, and XOR (you'll learn these in CS213)
  - Importantly, ^ is not exponentiation!!!

Adding and Subtracting one

- ++   --
  - Shorthand for plus 1 or minus 1
  - `++a` -> `a += 1` -> `a = a + 1`

- The auto-increment/decrement operators can go before or after the variable
  - (`--x`) subtracts one and returns the new value of x from the expression
  - (`x--`) subtracts one but returns the *old* value of x from the expression

  - Usually, this doesn't matter, unless you write complicated statements that combine assignment and conditions
  - `if (--x > 0)` … (please just never do this)

Implementing Fibonacci in C

fib.c

```
long fib_iterative(int n){
    long curr = 0;
    long next = 1;

    for (int i = 0; i < n; ++i) {  // i++ also works
        long prev = curr;

        curr = next;
        next = prev + curr;
    }

    return curr;
}
```

$$fib(n) = \begin{cases} n, & if\ n < 2; \\ fib(n-2) + fib(n-1), & otherwise \end{cases}$$

`typedef` can be used to make new C type names

- Typedef creates a new type name that is a copy of an existing type

- Typedef keyword is followed by two types
  - First type: the original type name
  - Second type: the new type name

- Example:

```
typedef int x_coordinate_t;
x_coordinate_t my_variable = 5;
```

Ternary Operator

- ? :
  - Shorthand version of an if statement, determining result of expression
  - Example:
    - `return (a < 5) ? a : b;`

    equivalent to

    - ```
      if (a < 5) {
        return a;
      } else {
        return b;
      }
      ```
  - You won't need to use this. Usually, it just makes code harder to read.
      ```

Outline

- Unix Shell
  - Navigation
  - Working with files
- Compilation
  - Separate Compilation
  - Makefiles
  - Pre-processor
- More C syntax
  - Computing Fibonacci Numbers
  - Iteration
  - Input and Output
  - Other C Syntax

Outline

- Bonus: these are optional extra things that you might be interested in
    - They won’t be on a quiz, but may come up in real code

- **More Pre-processor**

- Makefile Syntax

What else can the pre-processor do?

- Macros
  - Text substitutions made by the pre-processor
- Compile-time code inclusion
  - Determine which code is actually compiled based on flags
- Pragma
  - Special commands to the compiler

C macros

```
#define NAME_OF_MACRO value_of_macro
```

- Examples:

```
#define LENGTH 20
#define FAIL_MESSAGE "There was an error!\n"
```

- The pre-processor pastes the text of the “value” wherever it finds the macro “name” in the source code
  - Useful for defining values that will be used in code
  - Again, be careful about weird bugs here!

Macro functions

- Macros can be used as functions as well

```
#define DEBUG(msg) printf(msg)

#define MIN(a, b) ((a < b) ? a : b)
```

- Generally, avoid this
  - You could just write a C function to do the operation instead
    - And the compiler will check this for errors better

  - It can be tricky to get right

Example of macro function trickiness

```
#define ADD(a, b) a+b

int x = ADD(3,4)*5; // Expects 7*5=35
```

- The pre-processor will expand this to:

```
int x = 3+4*5; // Expects 7*5=35
```

- Extra parentheses around the macro value prevent this issue

```
#define ADD(a, b) (a+b)
```

Ifdef in C

- The pre-processor evaluates the statement before compilation and either includes or removes the text
  - Useful because the code literally does not exist if removed

```
#ifdef DEBUG
  printf("Debug message here\n");
#endif
```

- Ifdef hell: when you can't figure out which C code is actually being compiled due to too many `#ifdef`s

Pragma examples

- Pragmas tell the C compiler to do something
  - Turn on/off warnings
  - Various compiler tricks that are important for low-level OS code

- Most common example: `#pragma` once at the top of each header
  - Tells the compiler to track this file and only paste it in a given C file once
  - Otherwise could end up with a bunch of different copies

  - Old C code uses `#ifdef` at the top of header files for the same task
    - Paired with an `#endif` at the very bottom of the file

Outline

- Bonus: these are optional extra things that you might be interested in
  - They won't be on a quiz, but may come up in real code
- More Pre-processor
- **Makefile Syntax**

Bonus: Makefile for example_project/

example_project/

- Take a look at these if you want to understand the Makefile for the interact and posn_test programs from today's lecture files
  - In the example_project/ directory

Bonus: Makefile for building interact and posn_test

- These rules encode the dependency diagram from a few slides back (but with preprocessing and translation combined)

```
interact: interact.o posn.o
    cc -o interact interact.o posn.o

posn_test: posn_test.o posn.o
    cc -o posn_test posn_test.o posn.o

interact.o: interact.c posn.h
    cc -c -o interact.o interact.c

posn_test.o: posn_test.c posn.h
    cc -c -o posn_test.o posn_test.c

posn.o: posn.c posn.h
    cc -c -o posn.o posn.c
```

Bonus: Makefile for building interact and posn_test

- Good programmers are lazy and hate repetition. So much repetition here!

```
interact: interact.o posn.o
	cc -o interact interact.o posn.o

posn_test: posn_test.o posn.o
	cc -o posn_test posn_test.o posn.o

interact.o: interact.c posn.h
	cc -c -o interact.o interact.c

posn_test.o: posn_test.c posn.h
	cc -c -o posn_test.o posn_test.c

posn.o: posn.c posn.h
	cc -c -o posn.o posn.c
```

Bonus: Makefile for building interact and posn_test

- You don’t have to repeat the goal in each recipe
  - It’s better to use the special variable $@ instead

```
interact: interact.o posn.o
	cc -o $@ interact.o posn.o

posn_test: posn_test.o posn.o
	cc -o $@ posn_test.o posn.o

interact.o: interact.c posn.h
	cc -c -o $@ interact.c

posn_test.o: posn_test.c posn.h
	cc -c -o $@ posn_test.c

posn.o: posn.c posn.h
	cc -c -o $@ posn.c
```

Bonus: Makefile for building interact and posn_test

- Similarly, $^ is a variable that stands for the prerequisites
  - Or $< when you only want the *first* prerequisite

```
interact: interact.o posn.o
	cc -o $@ $^

posn_test: posn_test.o posn.o
	cc -o $@ $^

interact.o: interact.c posn.h
	cc -c -o $@ $<

posn_test.o: posn_test.c posn.h
	cc -c -o $@ $<

posn.o: posn.c posn.h
	cc -c -o $@ $<
```

Bonus: Makefile for building interact and posn_test

- Now note that the bottom three compilation rules are the same except for the filename. We can replace them with a pattern rule

```
interact: interact.o posn.o
	cc -o $@ $^

posn_test: posn_test.o posn.o
	cc -o $@ $^

interact.o: interact.c posn.h
	cc -c -o $@ $<

posn_test.o: posn_test.c posn.h
	cc -c -o $@ $<

posn.o: posn.c posn.h
	cc -c -o $@ $<
```

Bonus: Makefile for building interact and posn_test

- This pattern says we can build any .o file from a matching .c file

```
interact: interact.o posn.o
	cc -o $@ $^

posn_test: posn_test.o posn.o
	cc -o $@ $^

%.o: %.c posn.h
	cc -c -o $@ $<
```

Bonus: Makefile for building interact and posn_test

- That pattern is pretty generic except for the reliance on posn.h
  - Let's break that out into a separate rule

```
interact: interact.o posn.o
	cc -o $@ $^

posn_test: posn_test.o posn.o
	cc -o $@ $^

%.o: %.c
	cc -c -o $@ $<

interact.o posn_test.o posn.o: posn.h
```

Bonus: Makefile for building interact and posn_test

- And we really ought to make the compiler used a variable
  - Then others could change it out if desired

```
interact: interact.o posn.o
	$(CC) -o $@ $^

posn_test: posn_test.o posn.o
	$(CC) -o $@ $^

%.o: %.c
	$(CC) -c -o $@ $<

interact.o posn_test.o posn.o: posn.h
```

Bonus: Makefile for building interact and posn_test

- Finally, there are often compiler options we want to pass in
  - Here are the standard variables for holding those

```
interact: interact.o posn.o
	$(CC) -o $@ $^ $(CFLAGS) $(LDFLAGS)

posn_test: posn_test.o posn.o
	$(CC) -o $@ $^ $(CFLAGS) $(LDFLAGS)

%.o: %.c
	$(CC) -c -o $@ $< $(CPPFLAGS) $(CFLAGS)

interact.o posn_test.o posn.o: posn.h
```