Slide 1

Administrivia

- Reminder: Homework 4 due today.
- Homework 5 posted; due next Wednesday.

Slide 2

The `make` Utility

- Motivation: Most programming languages allow you to compile programs in pieces ("separate compilation"). This makes sense when working on a large program — when you change something, just recompile parts that are affected.
- Idea behind `make` — have computer figure out what needs to be recompiled and issue right commands to recompile it.
- (I think all of you have seen this, in CSCI 1120? but review.)

Slide 3

Makefiles

- First step in using `make` is to set up "makefile" describing how files that make up your program (source, object, executable, etc.) depend on each other and how to update the ones that are generated from others. Normally call this file `Makefile` or `makefile`.

  Simple example on "sample programs" page.
- When you type `make`, `make` figures out (based on files' timestamps) which files need to be recreated and how to recreate them.

Slide 4

Useful Command-Line Options

- `make` without parameters makes the first "target" in the makefile. `make foo` makes `foo`.
- `make -n` just tells you what commands would be executed — a "dry run".
- `make -f otherfile` uses `otherfile` as the makefile.

Slide 5

Defining Rules

- Define dependencies for a rule by giving, for each "target", list of files it depends on.
- Also give the list of commands to be used to recreate target.

NOTE!: Lines containing commands must start with a tab character. Alleged paraphrase from an article by Brian Kernighan on the origins of UNIX:

> The tab in makefile was one of my worst decisions, but I just wanted to do something quickly. By the time I wanted to change it, twelve (12) people were already using it, and I didn't want to disrupt so many people.

Slide 6

Phony Targets

- Normally targets are files to create (e.g., executables), but they don't have to be. So you can package up other things to do ...
- Example — many makefiles contain code to clean up, e.g.:

```
clean:
        -rm *.o main
```

To use — `make clean`.

Slide 7

Variables in Makefiles

- You can also define variables, e.g.:
  - List of object files needed to create an executable. Then use this list to specify dependencies, command.
  - Pathname for a command, options to be used for all compiles, etc.
- (Used in example.)

Slide 8

Predefined Implicit Rules

- `make` already knows how to "make" some things — e.g., `foo` or `foo.o` from `foo.c`.
- In applying these rules, it makes use of some variables, which you can override.
- A simple but useful makefile might just contain:

```
CFLAGS = -Wall -pedantic -O -std=c99
```

- This can also make longer makefiles not as long. (Revised example.)

Slide 9

`make` — Overriding Variables at Runtime

- Something else that can be useful in makefiles is providing variables that can be overridden at runtime. For example, if in the makefile you have

```
CFLAGS = -Wall -pedantic $(OPT)
OPT = -O
```

you can override `$OPT` with e.g., `make OPT=-g foo`.

Slide 10

Minute Essay

- Have you used `make` in other courses? (I seem to remember hearing that Dr. Fogarty uses it in Functional maybe?)