Slide 1

Administrivia

- Most people said they would be in class last Monday. Only three actually showed up. What happened?
- Reminder: Homework 9 due . . . Scheduled due date was today; extended to Wednesday.
- Reminder: Quiz 6 Wednesday. Likely topics are strings and arrays.
- Homework 10 on the Web. Due date shown as next Monday, but that will likely change to later.
- Be advised that the ACM tutors likely will not be available past the end of this week. I'm thinking of having some sort of "open lab" during the reading days?

Slide 2

User-written "Library" Code

- You know about calling functions in the C standard library (e.g., `printf`) in your code. One advantage of having a library is that this code only has to be written once, and then every program can use it.
- If you write more-complex programs, it may make sense to write your own "library" of functions to be called from more than one program, putting them in a separate source file. (Homework 10 is an example — a few functions that make up a "library", and two programs that use these functions.)

Slide 3

User-written "Library" Code, Continued

- How to "package" these library functions? At least two possibilities.
- One is to put them in a `.h` file and use `#include` to include it in every program that uses the functions.
- Another way (which is pretty much how the standard library functions are packaged) is to have a `.h` file containing declarations and a companion `.c` file with definitions. You then `#include` the `.h` file in programs that use the functions, and use "separate compilation" . . .

Slide 4

Building Large Programs — Separate Compilation

- For large programs it's often better to split up code into more than one source file — for readability if nothing else.
- How then to make the executable? A good way is to compile each `.c` file separately (with `gcc -c`) and then use `gcc` to "link" the resulting `.o` ("object code") files to produce the executable. (Note too that while `gcc` names the executable `a.out` by default, it will call it something else if you say to.)
- Sounds complicated? well, not as simple as compiling a single `.c` file, but . . .

Slide 5

A Little About `make`

- 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.
- (Caveat: `make` is a UNIXworld thing. I feel sure there's something analogous for people developing software under Windows but am not sure what it is!)

Slide 6

Makefiles

- First step in using `make` is to set up "makefile" with "rules" 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`.

  (Example: `Makefile.v1` in "makefile example" on sample programs page — rename to `Makefile` to easily use.)
- When you type `make`, `make` figures out (based on files' timestamps) which files need to be recreated and how to recreate them.

Slide 7

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 8

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 9

Variables in Makefiles

- You can also define variables; define with, e.g., `CFLAGS = -Wall` and reference as `$(CFLAGS)`.
- One good use is options to be used for all compiles.
- Another good use is to specify lists of files.

Slide 10

"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 11

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
```

Slide 12

Predefined Implicit Rules, Continued

- You can also make use of these predefined implicit rules to make your makefiles simpler.
- (Example: `Makefile.v2` in example.)

Slide 13

Minute Essay

- Questions? otherwise just sign in.