Slide 1

Administrivia

- (None?)

Slide 2

Minute Essay From Last Lecture

- Comments on homework 5 ranged from “harder but far more interesting” to “not that hard”.
- A few people commented on that second problem, and — one reason I assigned it was as an example of processing text input a character at a time, which is something I think is very useful and idiomatic in C but which maybe doesn’t occur to to people coming from languages in which it’s easy to read text input a line at a time.

Slide 3

A Little About the C Preprocessor

- C logically divides the process of producing an executable into distinct phases. First phase is "preprocessing".
- Preprocessing makes use of "preprocessor directives", which start with a `#`.
- Examples you've seen — `#include` to include information about library functions, `#define` to define constants.

Slide 4

A Little More About the C Preprocessor

- Other functionality includes macros and "conditional compilation":
- Macros can be used to do a primitive kind of generic programming (more on next slide).
- Conditional compilation often used to tailor library or other code to specific environments. Also allows writing `.h` files that can be included more than once without harm. Lots of examples in files in `/usr/include`.
- More in chapter 14, some beyond the scope of this course. Focus is on relatively simple text manipulation.

Slide 5

Macros in C

- Simple example (and a very typical use):

```
#define MAX(a, b) ((a) > (b) ? (a) : (b))
int x = MAX(10,20);
```

  Another use might be a `SQUARE` macro.

- More-complex macros can be used to generate multiple lines of code, though this can get (in my opinion) messy and not very readable.
- If you find yourself writing the same ones repeatedly, can put them in a file (typically with extension `.h`) and use `#include` (with filename in double quotes) to include them.

Slide 6

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.

Slide 7

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`.

  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 8

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 9

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 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

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.
- Example:

```
objs = main.o foo.o
CFLAGS = -Wall -pedantic -std=c99
main:   $(objs)
        gcc $(CFLAGS) -o main $(objs)
```

Slide 12

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
```

- Or you could use

```
CFLAGS = -Wall -pedantic -std=c99 $(OPT)
OPT = -O
```

and then optionally override the `-O` by saying, e.g., `make OPT=-g foo`.

Slide 13

Minute Essay

- Have you seen `make` in another course or elsewhere?
- (And best wishes for a good spring break!)