Slide 1

Administrivia

- Homework 5 to be on the Web tomorrow. Due in a week.

Slide 2

I/O in C — Review

- `getchar` and `putchar` provide simple character-at-a-time I/O to standard input/output.
- `printf` and `scanf` provide more sophisticated functionality, but again for standard input/output.
- I/O redirection provides one way to work with files. Is there something more general? Yes . . . .

Sidebar — Input/Output Redirection in UNIX/Linux

- In programming classes I talk about “reading from standard input” rather than “reading from the keyboard”, and “writing to standard output” (or “writing to standard error”) rather than “writing to the screen”.
- What’s the difference?

Slide 3

I/O Redirection, Continued

- `stdin` (standard input) can come from keyboard, file, or from another program or shell script.
- `stdout` and `stderr` (standard output, error) can go to terminal or file (overwrite or append), separately or together.

Slide 4

Slide 5

I/O Redirection, Continued

- For example — to redirect output of `ls` to `ls.out`, type

  `ls >ls.out`

  (Overwrites `ls.out`. To append, replace > with >>.)

  To also redirect any error messages, append `2>&1`.
- To redirect input, use `<infile`.

Slide 6

Streams

- C's notion of file I/O is based on the notion of a *stream* — a sequence of characters/bytes. Streams can be *text* (characters arranged into lines separated by something platform-dependent) or *binary* (any kind of bytes). UNIX/Linux doesn't make a distinction, but some other operating systems do.
- An input stream is a sequence of characters/bytes coming into your program (think of characters being typed at the console).
- An output stream is a sequence of characters/bytes produced by your program (think of characters being printed to the screen, including special characters such as the one for going to the next line).

Slide 7

Streams in C

- In C, streams are represented by the type `FILE *` — i.e., a pointer to a `FILE`, which is something defined in `stdio.h`.
- A few streams are predefined — `stdin` for standard input, `stdout` for standard output, `stderr`) for standard error (also output, but distinct from `stdout` so you can separate normal output from error messages if you want to).
- To create other streams . . .

Slide 8

Creating Streams in C

- To create a stream connected with a file — `fopen`.
- Parameters, from its `man` page:
  - First parameter is the name of the file, as a C string.
  - Second parameter is how we want to access the file – read or write, overwrite or append — plus a `b` for binary files, also a string.
  - Return value is a `FILE *` — a somewhat mysterious thing, but one we can pass to other functions. If `NULL`, the open did not succeed. (Can you think of reasons this might happen?)

Working With Streams in C

- To read from an input stream — `fscanf`, almost identical to `scanf`. To write to an output stream — `fprintf`, almost identical to `printf`. `fgetc` and `fputc` provide single-character input and output.
- When done with a stream, `fclose` to tidy up. (Particularly important for output files, which otherwise may not be completely written out.)

Slide 9

Reading Text Strings

- Getting text-string input is surprisingly tricky. `scanf` (or `fscanf`) seems like an obvious choice, but:
  - it can't read a string that includes blanks, and
  - it has no nice way to limit the number of characters read to the size of the array being read into.

  .
- Getting a whole line is probably better. `gets()` is an obvious/simple choice for reading from standard input, but it also has no way to limit how much is read. `fgets()` is better. (Look at its `man` page.)

  (Also notice `puts()` — simple way to write out a text string.)

Slide 10

Slide 11

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.
- Other functionality includes macros and "conditional compilation". More in chapter 14, some beyond the scope of this course. Focus is on relatively simple text manipulation.

Slide 12

A Little More About `gcc`

- Many, many compiler options for `gcc`. One of the most useful is `-Wall`.
- To automate using them every time, you can use the UNIX utility `make` ...

Slide 13

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 14

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 15

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

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

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

Slide 16

Minute Essay

- None — sign in.