Slide 1

Administrivia

- Next homework to be on the Web soon; due after the holiday. (As usual, I will send mail if it's up before next class.)
- See solution for Quiz 3 online for a comment about the code.

Slide 2

Input/Output Redirection in UNIX/Linux

- We talk about `scanf` reading input "from the user" or "from the keyboard", and `printf` printing "to the screen".
- But that's not quite right — really, `scanf` reads from *standard input*, and `printf` writes to *standard output*.
- What's the difference? can *redirect* standard input/output to use (text) files instead, or to have one program use as its input the output of another program ("pipes").

  NOTICE that the textbook does not mention this, saying instead that `scanf` always reads from the keyboard and `printf` always write to the screen. Maybe in some (other) environments??

Slide 3

Input/Output Redirection in UNIX/Linux, Examples

- Redirecting to use files as input/output:

  `myprogram < test1-in > test1-out`

  to have `myprogram` get its input from `test1-in` rather than the keyboard, and put its output in `test1-out` rather than showing it on the screen. (Overwrites `test1-out`. To append instead, use `>> test1-out`.)

  This is (part of) how I grade your programs!

- Redirecting to "pipe", to display output one screenful at a time and allow some searching:

  `myprogram | less`

Slide 4

Files and C

- Why files? You probably already know: Things stored in memory vanish when you turn the computer off; to preserve them, usually save them as *files*.
- We know one way for a C program to get its input from a file, or write its output to a file — I/O (input/output) redirection. But this makes it difficult to get input from more than one source, or save output in more than one place.
- So C (like many other programming languages) provides ways to work more generally with files.

Slide 5

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 doesn't make a distinction, but 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 6

Streams in C

- In C, streams are represented by the type `FILE *`. `FILE` is something defined in `stdio.h`. (As usual, the `*` means pointer — discussed a bit already, more soon.)
- 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 — next slide.

Slide 7

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.
  - Second parameter is how we want to access the file – read or write, overwrite or append — plus a `b` for binary files.
  - 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?)

Slide 8

Working With Streams in C

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

Slide 9

Minute Essay

- Anything interesting to report about Homework 5?