Slide 1

Administrivia

- Reminder: Homework 5 due today.
- Homework 6 on the Web; due in two weeks.

Slide 2

Minute Essay From Last Lecture

- About Homework 4, many people found it more difficult than previous assignments, but several said it was also more interesting. I agree!
- A few people mentioned learning that arrays in C aren't initialized by default. (As with so many things, probably meant as an efficiency improvement.)
- Several mentioned trouble converting math to code in the first problem. And in working with individual students in lab it seems that several read the subscripted variable r_{n-1} as $(r_n) - 1$. That was a new-to-me problem and — "hm!"?
- Several people mentioned having trouble understanding the second problem. Students have had this problem in the past, and I may owe you an apology for not trying to revise and clarify.

Slide 3

Character-Oriented I/O in C

- Two useful functions to know about: `getchar` and `putchar`.
- Both treat characters as integers (which is allowed). `getchar` returns a special value, `EOF`, at "end of file". How to signal this when standard input is from keyboard is system-dependent — often(?) control-D on UNIX-like systems.
- (Sample program `echo-text.c` illustrates using these — not shown in class.)

Slide 4

I/O in C — Recap

- `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.
- Reading text strings *safely* is surprisingly difficult, so I say when you *can* read text a character at a time it may make sense to do so (as in one of the problems on Homework 6).
- I/O redirection provides one way to work with files. Is there something more general? Yes. ("Of course"?)

Slide 5

File I/O — 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 6

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`.

  (`FILE` is an example of an "opaque data type" — something defined in a library, the details of which might vary among implementations and which should not matter to users.)
- 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 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, 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?)

Slide 8

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

- As noted previously, getting text-string input is surprisingly tricky. `scanf` (or `fscanf`) seems like an obvious choice, but it has limitations. Getting a whole line is probably better, and for that `fgets()` is the better choice.
- Because of this, I much prefer to pass such things as filenames as command-line arguments.

Slide 10

Simple Examples

- First do a simple example of character-oriented I/O, using `getchar` and `putchar` for a first version and then `fgetc` and `fputc`.
- Then try an example (a revised program to sum inputs) of using `fscanf` and `fprintf` to read/write integers. Notice that `fscanf` "fails" in two situations: end of file and bad input. One way to tell which has happened is with `feof()`, which returns "true" at EOF. *Notice* that this function only returns "true" *after* you've tried to read something but EOF was detected. (Some published examples get this wrong!)

Slide 11

Minute Essay

- Anything noteworthy about Homework 5 (interesting, difficult, etc.)?
- How has the pace/workload of this class been so far? do you feel like it's about right for a one-unit course (which is supposed to represent about three hours of work per week, in and out of class)?