Slide 1

Administrivia

- Reminder: Homework 4 due today.
- Homework 5 on the Web. Due in two weeks. Many parts make use of file I/O, which is next week's topic, but . . .
- One problem asks you to fill in the body of a function to sort an array, and you should be able to do that.
- You could probably start looking at the other problems as well and possibly do some parts of it based on what we do today, and put off the parts that involve working with files until next week.

Slide 2

Minor C Programming Tip

- In some of the sample programs, `main()` returns `EXIT_SUCCESS` or `EXIT_FAILURE`. These are constants defined in `stdlib.h` and somewhat more guaranteed to be more portable than 0 or 1 (not to mention that they make it clearer what's being done?).

Slide 3

Pointers, Characters, and Strings in C — Review

- Pointers are, roughly speaking, memory addresses. Useful in many contexts. If they don't make sense to you yet, practice using them may help?
- Characters in C are small integers, big enough to represent ASCII characters but possibly not much more.
- Strings in C are arrays of characters, ending with "null character" (`\0`). Can operate on them as arrays or using pointers.
- (Example — several ways to write a "string length" function.)

Slide 4

Characters and Strings in C — Library Functions

- C's standard library is pretty limited but does contain some useful functions for operating on character/string data.
- Some useful ones are `isdigit` etc. for characters and `strlen`, `strcmp`, `strchr`, and `strstr` for strings.

  (Notice that you need `strcmp` to compare strings for equality; == compares *pointers* so generally will not do what you want.)

Slide 5

Strings in C — Pitfalls

- Most functions assume that strings are properly terminated. (What do you think happens if they're not?)
- Many functions that store into a string have no way to check that it's big enough.

  So getting text input from standard input *safely* is surprisingly difficult! `scanf` can be made to check, but not (in my opinion) nicely, and it stops on whitespace anyway. `gets` gets a full line, but notice what `gcc` says when you use it. `fgets` is maybe better but has its limitations too.

Slide 6

Another Way to Get Input — Command-Line Arguments

- Now that we know about arrays, pointers, and text strings, we can talk about command-line arguments. What are they? text that comes after the name of the program on the command line (e.g., when you write `gcc -Wall myprogram.c`, there are are two command-line arguments), possibly modified by the shell (e.g., for filename wildcards).
- Most programming languages provide a way to access this text, often via some sort of argument to the main function/method.

Slide 7

Command-Line Arguments in C

- In C, command-line arguments are passed to `main` as an array of text strings. So if you define `main` as

  ```
  int main(int argc, char * argv[]) { .... }
  ```

  `argc` is the number of arguments, plus one, and `argv` is an array of strings containing the arguments.

  ("Plus one"? yes, `argv[0]` is something system-dependent, often the path for the program's executable.)

  (Example — simple program to echo command-line arguments.)

- What if you want to get numeric input? you must convert string pointed to by `argv[i]` to the type you want (more shortly).

Slide 8

Command-Line Arguments and UNIX Shells

- Be aware that most UNIX shells do some preliminary parsing and conversion of what you type — e.g., splitting it up into "words", expanding wildcards, etc., etc.
- If you don't want that — enclose in quotation marks or use escape character (backslash).

Slide 9

Converting Strings to Numbers

- As noted, command-line arguments are strings. Two sets of functions for converting.
- One (`atoi` etc.) is easy to use but does no error checking (so I say avoid).
- Other (`strtol` etc.) is more trouble but does let you check for errors.

(Improve echo program.)

Slide 10

Minute Essay

- Anything noteworthy about Homework 4 (interesting, difficult, etc.)?