Slide 1

Administrivia

- Homework 4 will be on the Web later today / early tomorrow (I will send mail). Due next Monday.

Slide 2

Minute Essays

- What was interesting/difficult about Homework 2? one person mentioned appreciating what library functions do for you; several mentioned things related to new and lower-level language.
- What was interesting/difficult about Homework 3? recursion ("might just write it in Scala and translate" — not the worst idea but perhaps not optimal?), memoization.
- Memoization — is this a cache? sort of!
- Are C arrays linked lists? No.

Slide 3

Strings in C

- Many languages have nice ways of working with text (character strings). What C provides is — no surprise — somewhat primitive.
- In C, strings are arrays of `char`s, with the convention that the actual text of interest is followed by a null character (8-bit zero, represented in code as `'\0'`.

Slide 4

Working with Strings in C

- You can operate on individual characters however you see fit (accessing them as elements of the array). Or you can access them using pointers to `char`. (Recall that arrays and pointers are interchangeable in most contexts.)
- There are some useful standard-library functions for working with characters; `man ctype.h` will list them.
- There are also standard library functions for some common operations (e.g., `strcmp` to compare two strings — returns -1/0/1 depending on which string is lexicographically first). Simplest way to find them may be `man -k string` and ignore everything but the last few screenfuls.
- `scanf` and `printf` use `%s` to read/write strings. (Use with caution — next slide.)

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.

Slide 6

Sidebar: Basics of 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.
- More about input/output soon.

Slide 7

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 8

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.)
- What if you want to get numeric input? you must convert string pointed to by `argv[i]` to the type you want, e.g., with `atoi` or `strtol`.

Slide 9

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 10

Simple Examples

- Program to echo command-line arguments and do some simple things with them.

Minute Essay

- TBA