Slide 1

Administrivia

- Homework 7 (last one) to be on Web later today; due next Tuesday.

Slide 2

Pointers — Review

- Most/many programming languages provide a way to "point to" something in memory (such as a variable). In C, these are called pointers, and you declare them by putting a `*` after the type of the thing pointed to. (Notice that this means you can have pointers to pointers!)
- You can get the address of a variable with `&`. You "dereference" a pointer (access what it points to) with `*`.
- One important use of pointers is to allow returning more than one thing from a function, as `scanf` can.

Slide 3

Pointers, Arrays, and Pointer Arithmetic in C

- C treats pointers and arrays as interchangeable in most respects. (This is why it works that many functions whose parameters are supposed to be strings — arrays of characters — declare them as pointers. `fopen` is an example.)
- C also permits doing some arithmetic operations on pointers (addition and subtraction). Adding n to a pointer that points to *type* advances it n times the size of *type*.

  Example: If `a` is an array of `int`s, `a[2]` and `*(a+2)` are equivalent. (This means we could write loops over arrays using pointers. Once upon a time that was sometimes more efficient. With current compilers, probably not so, so use whatever is most readable.)

Slide 4

Dynamic Memory and C

- With the old C standard, you had to decide when you compiled the program how big to make things, particularly arrays — a signficant limitation.
- Variable-length arrays help with that, but don't solve all related problems:

  In most implementations, space is obtained for them on "the stack", an area of memory that's limited in size.

  You can return a pointer from a function, *but* not to one of the function's local variables (because these local variables cease to exist when you return from the function).

Slide 5

Dynamic Memory and C

- "Dynamic allocation" of memory gets around these limitations — allows us to request memory of whatever size we want (well, up to limitations on total memory the program can use) and have it stick around until we give it back to the system.

  (The trick here is that most implementations differentiate between two areas of memory, a "stack" used for local variables, and a "heap" used for dynamic memory allocation. Usually the former is more limited in size.)

- To request memory, use `malloc`. To return it to the system, use `free`. (For short simple programs you can not bother with `free`, but for longer and more complicated programs, you should clean up when you can, or eventually you may run out of memory.)

Slide 6

Dynamic Memory and C, Continued

- Examples:

```
int * nums = malloc(sizeof(int) * 100);
char * some_text = malloc(sizeof(char) *
20);
free(nums);
```

- Book recommends "casting" value returned by `malloc`. Other references recommend the opposite! But you should check the value — if `NULL`, system was not able to get that much memory.

- (Redo sort/search example using dynamically allocated memory.)

Slide 7

Text Strings in C, Revisited

- As mentioned briefly last time: C represents text strings as arrays of characters, with the end of the string indicated by a special "null" character.
- There are many library functions useful for working with strings. But as practice working with arrays and pointers and dynamic memory, we could write some of our own . . .

Slide 8

Minute Essay

- None — quiz.