Slide 1

Administrivia

- Reminder: Quiz 4 Monday. (If you cannot reasonably be present then, talk to me about whether it's possible to make up this quiz later.) Likely topics include material from this week and the little we did about sorting and "order of magnitude" of algorithms. (So, review how the three sorting algorithms in the textbook work, focusing on how they move data around rather than on code.)
- Next homework still "in work" . . .

Slide 2

Pointers Revisited

- Every time you call `scanf`, you pass it at least one parameter of the form `&x`. What does that mean? Also, when you look at `man` pages for some functions, they show function declarations with parameters of the form *type* `*`. What does that mean?
- To explain, we need one more kind of variable — *pointers*. A pointer, as its name suggests, points to something — namely, a location in memory. Typically a pointer "points to" a variable.

Slide 3

Pointers in C

- Many programming languages provide something like pointers. Unlike some more-recent languages, C allows you to have both pointer variables and non-pointer variables.
- To a first approximation, C pointers are just memory addresses — i.e., numbers — but they are declared to point to variables (or data) of a particular type. Example:

```
int * pointer_to_int;
double * pointer_to_double;
```

Slide 4

Pointers in C — Operators

- & gets a pointer to something in memory. So for example you could write

```
int x;
int * x_ptr = &x;
```

- * "dereferences" a pointer. So for example you could change X above by writing

```
*x_ptr = 10;
```

Slide 5

Pass By Reference, Sort Of — Review(?)

- Functions can only explicitly return a single value — a significant limitation. Pointers provide a way to get around that: By passing a pointer to something, rather than the thing itself, can in effect have a function return multiple things.
- To make this work, declare the function's parameters as pointers, and pass addresses of variables rather than variables. (This is how `scanf` does what it does, and why you need the &.)
- (The "sort of" in the slide title is because this is not true pass by reference as in, e.g., C++, but the effect is the same.)
- (Example.)

Slide 6

A Little About Strings in C — Review(?)

- Most programming languages provide a way to represent text (sequence of characters). C differs from some others you might use in providing only a very simple way that exposes details of the implementation.
- In C, a (character) string is an array of characters, with a *null character* (written `'\0'`) at the end. So to declare a variable to hold a string, you might write

```
char mystring[100];
```

- Character literals written with single quotes, string literals with double quotes.

Slide 7

Strings in C, Continued

- To print a string, can use `%s` with `printf`, or `puts`.
- To read a string from standard input or a file, can use `scanf` with `%s` — but this is risky unless you limit how many characters are read! May be better to read a whole line with `fgets`, but that also has a downside (must deal with end-of-line character).

Slide 8

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. Many many examples . . .)
- 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. (So 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 9

Working With Text Strings in C

- Nany library functions useful for working with strings.
- Significant problem in working with strings — no natural maximum size, so must decide how big to make the array of characters used to hold one — and then be sure you don't try to put in too many characters.
- Some library functions let you say how big the array is; some don't. *Always* be as careful as you can when working with strings; trying to store a string in an array not big enough is a source of "buffer overflows", which can lead to program crashes and more subtle problems, including security risks.

Slide 10

Working With Text Strings in C, Continued

- Many library functions for working with strings use/return pointers. Pointer arithmetic allows for some interesting uses of these functions.
- (Examples as time permits.)

Slide 11

Minute Essay

- Questions? otherwise just sign in.