Slide 1

Administrivia

- Homework 5 will be on the Web soon. Due next week.

Slide 2

Function Pointers

- You know from more-abstract languages that there are situations in which it's useful to have method parameters that are essentially code. Some languages make that easy (functions are "first-class objects") and others don't, but almost all of them provide some way to do it, since it's so useful — e.g., providing a "less-than" function for a generic sort.
- In C, you do this by explicitly passing a pointer to the function.

Slide 3

Function Pointers in C

- The type of a function pointer includes information about the number and types of parameters, plus the return type.
- Example — last parameter to library function `qsort` (in its `man` page). Call this by providing, in your code, a function with declaration

```
int my_compare(const char *, const char *);
```

and using `my_compare` as the last parameter to `qsort`.

(Revise sort example to try this.)

Slide 4

Pointers Versus Arrays — One More Thing

- We've said that pointers and arrays are in most contexts equivalent. One potential benefit of this is that it can make it easier to work with substrings, subarrays, etc.
- Example — one more revision of the string-length example, using recursion.

Slide 5

User-Defined Types, Review/Recap

- Last time we discussed various kinds of "user-defined" types (`typedef`, `struct`, etc.).
- As an example, let's write code to implement a singly-linked list. This is also an example of how one might implement, at a low level, one of the nice abstractions (lists) found in higher-level languages.
- But first — we may end up with more code than will comfortably fit into one file, so a bit more about compiling . . .

Slide 6

A Little More About `gcc`

- Many, many compiler options for `gcc`. One of the most useful is `-Wall`.
- To automate using them every time, you can use the UNIX utility `make` . . .

Slide 7

A Little About `make`

- Motivation: Most programming languages allow you to compile programs in pieces ("separate compilation"). This makes sense when working on a large program — when you change something, just recompile parts that are affected.
- Idea behind `make` — have computer figure out what needs to be recompiled and issue right commands to recompile it.

Slide 8

Makefiles

- First step in using `make` is to set up "makefile" describing how files that make up your program (source, object, executable, etc.) depend on each other and how to update the ones that are generated from others. Normally call this file `Makefile` or `makefile`.

  Simple example (assuming `main.c` `#includes` `defs.h` and `foo.h`):

```
main:   main.o foo.o
        gcc -o main main.o foo.o
main.o: main.c defs.h foo.h
        gcc -c main.c
foo.o:  foo.c
        gcc -c foo.c
```

- When you type `make`, `make` figures out (based on files' timestamps) which files need to be recreated and how to recreate them.

Slide 9

Predefined Implicit Rules

- `make` already knows how to "make" some things — e.g., `foo` or `foo.o` from `foo.c`.
- In applying these rules, it makes use of some variables, which you can override.
- A simple but useful makefile might just contain:

```
CFLAGS = -Wall -pedantic -O -std=c99
```

- Or you could use

```
OPT = -O
CFLAGS = -Wall -pedantic -std=c99 $(OPT)
```

  and then optionally override the `-O` by saying, e.g., `make OPT=-g foo`.

Slide 10

Example — Singly-Linked List

- Consider code for a singly-linked list of integers, using two `struct`s, one for list nodes and one for the list itself.
- To make things more interesting, we could write all the functions to use recursion.
- (Start looking at code.)

Slide 11

Minute Essay

- None — sign in.