Slide 1

Administrivia

- Quiz 1 sample solution available online. Grades rather disappointing. However, only 10 points, and I do drop the lowest quiz score.
- Reminder: Homework 3 due Friday.

Slide 2

Sidebar: "Is This Answer Reasonable?"

- I'm inclined to think (hope?) that in other problem-solving courses you've heard that it's good, when you come with an answer, to ask whether it even makes sense. This course is no exception, though perhaps the principle is not as broadly applicable.
- I mention this because some people lost points on the quiz question about binary etc. numbers with answers that seem to me not to pass this test:
  - If you're converting an even number to binary, can the last bit ever be 1?
  - If you convert from one base to another, could the result be the same in both bases? (Yes, there are cases, but — what?)

Slide 3

Functions in C — Recap/Review

- Functions in C (as in other programming languages) are a way to break up a big problem into more manageable pieces and also to avoid duplication of code/effort.
- In C, unlike in many other languages, before you can call a function the compiler has to know about it. Two ways to do this:
  - Put the definition of the function before the call to it. Okay for short functions but not a great way to organize a program in general ("top-down" idea. And obviously(?) won't work for library functions.
  - Include a "function declaration" describing the function's parameters and return type before the call. The actual definition (code) can then be later in the file or in some other file. That line `#include <stdio.h>`? it includes declarations for `scanf`, `printf`, etc.
- ("Find roots" example continued.)

Slide 4

Functions and Recursion

- Something else we want to be able to do is repeat something some fixed number of times, or until some condition is true. (Examples include iterating over a large collection of values, or until some kind of convergence is reached.)
- We'll talk soon about some new constructs to do that soon, but we can do it now, with *recursion* — having a function call itself.

Slide 5

Recursion — Concepts

- Recursive function is one that calls itself.
- Obviously to make this work we need a way to stop recursing — a *base case* — otherwise we have something akin to the in-joke definition of GNU ("GNU is Not Unix").
- Also we need to be sure that every recursive call brings us closer to a base case.

Slide 6

Recursion — Implementation

- How it works: When you call any function, the current "state" (values of variables) is preserved ("pushed onto a stack"), and space is reserved for the called function's local variables (including parameters). When the function returns, this space is freed up again. So if we stack up recursive calls to the same function, each has its own copy of all local variables.
- Simple examples — factorial, Fibonacci numbers, counting(?), sum from input.

Minute Essay

- Have you encountered recursion previously? perhaps in a math class? how about proofs by induction (sort of a similar idea)?
- How did Quiz 1 compare to your expectations (length, topics, difficulty, etc.)? If you lost points, what do you think went wrong?