Slide 1

Administrivia

- Homework 4 on Web; due next Thursday.

Slide 2

Text Editors, Revisited

- As we said much earlier, source code for C programs is just text, and it can be written using any program (“text editor”) for creating and modifying text files.
- Also as mentioned much earlier, there are lots of text editors. If you don’t like `vim` — ask me about specific “how do I . . . ”, and/or try its graphical version `gvim`, and then experiment with others. Another instructor for this course uses `pico`.

Slide 3

Functions — Recap

- Purpose of functions — decompose problem into smaller problems. Also helps avoid duplicating code.
- C functions are similar in some ways to math functions, but can have side effects. (Sometimes the side effects are actually the only effects we care about — e.g., with `printf`).
- In C, parameters are passed *by value* — i.e., copied. This means that any changes made in a called function aren't visible to the caller, and also (apparently) that a function can only pass information back to its caller through its single return value.

  ("Apparently" is because there *is* a way, discussed in the textbook and used by `scanf`. We'll talk about it later, when we discuss pointers.)

Slide 4

C Functions — Declaration Versus Definition

- So far we've looked at function *definitions*, which consist of
  - A name.
  - Zero or more inputs (parameters).
  - A return type.
  - Some code to be executed when the function is called.
- In the hypotenuse example from last week, we defined a function `hypotenuse` and used it in `main`. How did the compiler know what we meant when we used `hypotenuse`? because we had a definition there in the file, earlier.

  How does the compiler know what we mean when we call, say, `printf`?

Sidebar — Compiling and Linking, Recap

Slide 5

- Early in the semester we talked about source code versus object code versus executables. Reviewing:
- Compilers read source code and produce object code, consisting of machine-language instructions and information about functions defined, functions needed.
- Linkers take these files and combine them into executables, complete programs that can be launched by the environment (operating system). Usually, this involves combining object code for your functions with system/library object code.

  (Caveat: Some systems also allow for library code to be brought in at runtime — “shared libraries” in Unix-speak, “DLLs (dynamic link libraries)” in Windows-speak.)

C Functions — Declaration Versus Definition, Continued

Slide 6

- “How does it know what we mean?” has two parts:
  - Compiler needs to know about the function’s parameters (how many, their types) and return types. It will make guesses if it doesn’t know, but it might guess wrong.
  - Linker has to be able to find function’s code.
- Compiler can get what it needs if we include a *function declaration* before the first use of the function.
- Linker can get what it needs if the function is also defined in the same file as its caller, *or* if it can find it in a library of compiled code.
- Example — revise hypotenuse program to have separate declaration and definition.
- Now think about `printf` again . . .

Slide 7

C Library Functions, Revisited

- The compiler gets what it needs to know about library functions from declarations in files included with an `#include` directive. There are standard places to find these files; `stdio.h` is in `/usr/include`. (Look at it briefly.) You can also tell the compiler other places to look.
- Where does the linker find the actual code? There's a list of standard places it looks, and some default files it looks at there. (For `printf`, `/usr/lib/libc.a` or `/usr/lib/libc.so`.) You can also tell it to look in other places, and/or at additional files. (That's what the `-lm` flag does — tells the linker to also look in the math library file.)

Slide 8

Using Functions Effectively

- Functions are most helpful for two purposes: decomposing the problem into manageable chunks, and avoiding duplication of code.
- Let's do a short example — a program that lets us convert several kind of "English" units (feet, inches, etc.) to metric equivalents. This can also be an example of using a character variable and the `switch` construct.

Functions and Recursion

Slide 9

- Something else we want to be able to do is repeat something some fixed number of times, or until some condition is true — for example, in the converter program, prompt again if we get invalid input.
- We'll talk next week about some new constructs to do that, but we can do it now, with *recursion* — having a function call itself.
- (Simple examples as time permits.)

Minute Essay

Slide 10

- How did the midterm compare to what you expected? with regard to length, difficulty, topics . . .