Slide 1

Administrivia

- Reminder: Homework 1 due Wednesday (11:59pm).

Slide 2

Minute Essay From Last Lecture

- No clear consensus w.r.t. “too fast” versus “about right”. I’ll try to slow down a bit.

Slide 3

"Hello World" Program Revisited

- Look again at the program we wrote in class previously. Most of it is standard boilerplate, to be discussed further soon. Single line you should pay attention to now is the one with `printf`.
- Goal for today — describe how to extend this to get input from "standard input" (keyboard by default), do simple computing, write results to "standard output" (terminal window by default).
- (Maybe worth pointing out now that this is the style of programs we'll write — simple text-mode user interface. This is really all you can do in standard C.)

Slide 4

Sidebar: C Design Goals

- Many currently-popular languages are big and complicated and designed to make the programmer's job easier. Often they include huge "libraries" to support interesting features such as GUIs, graphics, network communication, etc., etc.
- C, in contrast, was intended to be efficiently implementable on a very wide range of "platforms" (combination of hardware and operating system). It's therefore somewhat minimalist. GUIs, graphics, etc., can all be done from C, but only by using libraries that aren't part of standard C. And efficiency takes precedence over programmer convenience.

Slide 5

Variables in C

- In C as in most/many other programming languages, you need temporary storage for data — e.g. someplace to save an input value and/or intermediate results. For this we use *variables*.
- In C variables must be *declared*, each with both a name and a *type*. Effect of declaring a variable is to reserve memory (RAM) for a value of the specified type and give it a name that can be referenced. (Similar to Matlab, except for choice of types?) What a name can look like is somewhat restricted (see textbook).
- Types in C are pretty basic — integers, "floating-point numbers" (numbers with a fractional part), and characters. Integer types are represented as fixed-size binary numbers and come in various sizes. More about the others later.

Slide 6

Variables in C

- Variables are given values by *assignment statements* (using =, which here means "assign value on right to variable on left" rather than equality as in math!).
- Okay to change value with repeated assignments.

Slide 7

Expressions in C

- What's on the right side of an assignment — *expression*.
- Expressions in C are similar to those in math, with some differences/extensions, partly due to limited range of symbols and partly due to how hardware usually works:

  `*` and `/` for multiplication and division; on integers division produces *quotient only*; to get remainder use `%`.
- An expression has a *value*, which is determined by *evaluating* it. Evaluation may have *side effects* — e.g., `printf("hello\n")` is an expression, with the side effect of "printing" and a value that's usually ignored.

Slide 8

Assignment Statements Revisited

- Simplest programs are often basically a sequence of assignment statements (plus some "statements" that are really just expressions, such as that `printf` in the "hello world"program).
- Unless otherwise indicated, statements are executed in the order in which they appear in the code. (Sequential-ness is important and sometimes trips up beginners.)

Slide 9

Simple Output in C

- Use `printf` to display predefined text and values of variables.
- Syntax is that of "function call" (more later) with first parameter a "format string" that may include "conversion specifications", followed by zero or more expressions, one for each conversion specification.
- When statement is executed, expressions are evaluated and the results turned into something printable using those conversion specifications.

Slide 10

Simple Output in C — Conversion Specifications

- Conversion specifications say what kind of data is to be printed (integer, floating-point, etc.) and how.
- For example, `%d` prints an integer in base 10, `%x` prints an integer in base 16. Also options to print with a fixed width (so output lines up in columns), control number of digits after the decimal point, etc.
- `man 3 printf` for all the details. (There are a lot.) (What's that "3"? There are several things called `printf`; the 3 says we want the C library function.)

Slide 11

Simple Input in C

- Use `scanf` to get input. (Caveat: It has limitations and annoyances, but it's what almost all intro texts use and is simple. Doing a really great job of interactive input is surprisingly(?) difficult, especially in C, so we'll aim just to do a reasonable job.)
- Syntax very similar to that of `printf` except that rather than expressions you have *pointers* that say where to store value(s). More about pointers later; in this context, name of variable preceded by `&`.

Slide 12

Simple Examples

- Recap from last time: Compile (and link) with `gcc` I recommend *ALWAYS ALWAYS* compiling with optional flag `-Wall` so you get most optional warnings — sometimes annoying, but often very helpful! Example

  `gcc -Wall hello.c)`

  Then execute with `./a.out`.

  (Actually there are some other flags you should probably use too. More about that later.)
- (Simple example(s).)

Slide 13

Example — "Counting Change"

- Problem statement: Given a number of pennies, show how to represent it with minimum number of coins (pennies, nickels, etc.).
- First think about how you'd do this yourself. Then turn it into code.

Slide 14

Minute Essay

- Any questions? How similar is all of this to something you've used before, such as Matlab?