Slide 1

Administrivia

- I had planned to talk about binary numbers today and start on C next week, but — defer the former for now. Readings on "lecture topics and assignments" page revised accordingly.
- File access revisited: What we (the admins) intended was to configure things so that by default only the owner can access files. But in fact what we did only works for "login shells" such as you get with remote login. In a terminal window you get what I think is the default for this version of Linux, which is far more open.

  To fix this, edit your `.bashrc` file, as described last time (remember to save a copy first!) and put in the line

  ```
  umask 077
  ```

  and all should be well.

Slide 2

Minute Essay From Last Lecture

- One person mentioned looking forward to being able to write more complicated programs "that accomplish things that may be helpful for human life". Probably won't happen any time soon, alas — this is a beginning course — but if you can be patient I'd like to think that by semester end you'll be doing things that aren't totally trivial.
- A couple of people asked about what would be on exams or how to study. This being a course in programming, and programming being a form of problem-solving, most of what you do outside class will focus on your learning this type of problem-solving, by solving problems (writing programs). Exams will also focus on problem solving. (And they'll be open-book/open-notes. No memorization!)

Slide 3

Programming Basics and C

- First lecture describes relationship between what humans write ("source code") and what computers execute ("machine language").
- For C, usual process is that you write source code, and then it must be transformed not just into machine language, but into a complete "executable file" (machine language for your code, plus machine language for any library functions, plus information so operating system can load it into RAM and start it up). (Detail: This is for "hosted environment"; in some environments in which C is used, there may be no o/s.)
- So, what happens to your code . . .

  First it's "compiled" into "object code" (machine language).

  Then it's "linked" with any library object code to form "executable file". Sometimes this happens more or less invisibly when you run command to compile.

Slide 4

"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).

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*.
- Again in C as in many others — *variables*. In C variables must be *declared*, each with both a name and a *type*. Effect of declaring a variable is to reserve 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" (for now, real numbers), and characters. Integer types are represented as fixed-size binary numbers and include 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, and 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 often is not used.

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.

Slide 9

Simple I/O 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.
- Use `scanf` to get input. (It's not really very good for interactive programs, but it's what almost all intro texts use, so we will too, but keep in mind that it has limitations and annoyances.) 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; for now usually name of variable preceded by `&`.

Slide 10

Simple Examples

- Recap from last time: Compile (and link) with `gcc` I recommend *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`.
- (Examples as time permits. At least some of these will show up on "sample programs" Web page after class.)

Slide 11

Example — "Counting Change"

- Problem statement: Given a number of pennies, show how to represent it with minimum number of coins (pennies, nickels, etc.).
- First define the problem, possibly doing some examples without a computer. Might be a good time to also come up with a short list of sample inputs/outputs that can be used for testing later.
- Next figure out a strategy for solving it using the tools you have.
- Finally turn that into source code. Good idea to start by writing *comments*, because . . .

  When writing source code you are writing for two audiences! the compiler, yes, but also (usually) for human readers.
- (To be continued?)

Slide 12

Minute Essay

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