Slide 1

Administrivia

- (None.)

Slide 2

Programming Basics (as described in CSCI 1320)

- What computers actually execute is *machine language* — binary numbers each representing one primitive operation. Once upon a time, people programmed by writing machine language (!).
- Nowadays, "programming" as we will use it means writing *source code* in a *high-level language*. Source code is simply plain text, which . . . At this point we diverge from the explanation for beginners. Exactly what happens to get from source code to something the computer can execute varies among languages . . .

Slide 3

From Source Code to — What?

- Some high-level languages (such as the language understood by typical UNIX/Linux command shells) are directly interpreted by some other program.
- Others are *compiled* into *object code* (machine language) and then *linked* with other object code (including system libraries) to form an *executable* (something the operating system can execute).
- Java takes a somewhat intermediate approach — it's initially compiled into *byte code* (object code for a made-up processor), which is (in principle) interpreted by the runtime system (Java Virtual Machine), with system library code brought in at runtime. (In practice, often a "just-in-time" compiler translates byte code into native object code on the fly.)

Slide 4

Why Learn C? (For Java Programmers)

- Java provides a programming that's nice in many ways — lots of safety checks, nice features, extensive standard library. But it hides a lot about how hardware actually works.
- C, in contrast, has been called "high-level assembly language" — so it seems primitive in some ways compared to Java. What you get (we think!) in return for the annoyances is more understanding of hardware — and if you do low-level work (e.g., operating systems, embedded systems), it may well be in C.

Slide 5

Structure of a C Program

- Pre-processor directives: These begin with # and are used to (among other things) include in the compilation process information about libraries.
- Global identifiers (functions and variables). Function declarations here are often useful; variables are usually bad practice.
- Function(s), possibly containing variables, returning values, etc. Every complete program has exactly one `main` function.
- Most syntax should look familiar to Java programmers (no accident — Java was designed that way). Biggest exception may be what's not there — classes and exceptions being the most notable.

Slide 6

A Few Words About "Old C" Versus "New C"

- First ANSI standard for C — 1989. Widely adopted, but has some annoying limitations.
- Later standard — 1999. Many features are widely implemented, but few compilers support the full standard, and older programs (and some programmers concerned about maximum portability) don't use new features. Much of what we do in this class will focus on older standard for this reason.

Slide 7

A First C Program

- Let's write the traditional "hello world" program in C, using `vi`.

  (This tradition of having one's first program in a language print "hello world"? It comes from the early and still fairly authoritative book *The C Programming Language*, by Kernighan and Ritchie.)
- Once it's written, compile-and-link by typing `gcc hello.c`. (There are other options you should use, but for now this is okay.) Result is `a.out`.
- Execute by typing `a.out`.
- Now let's look at the program line by line . . .

Slide 8

Variables in C

- Simple variables (numbers, characters, etc.) are fairly similar to Java primitive variables. Key differences:
  - Sizes of numeric types aren't as strictly defined — e.g., a Java `int` is exactly 32 bits, but a C `int` may be more. (Why? to allow implementations to use whatever is most efficient.)
  - No `boolean` in C89.
  - `char` is an ASCII (not Unicode) character.
  - Integer types can be signed or unsigned.
- Arrays syntactically similar to Java, but more primitive (more about them later).
- Pointers similar to Java references, but more flexible / less safe.

Slide 9

Expressions, Statements, and Control Structures

- Most syntax is similar to Java (which is no accident) — within each function, code is organized into statements, which may contain expressions.
- Control structures are mostly the same as in Java — `if` , `while`, `do`, `switch`, `for`, etc. C doesn't have the simpler/newer form of `for` (referred to as "foreach").
- Key difference is the lack of classes (and supporting syntax), and use of pointers rather than references.

Slide 10

Functions

- Functions also are similar to those in Java, with a couple of key distinctions:
  - They have to be declared (or defined) before being referenced.
  - Pass-by-value semantics for parameters means you need pointers if you want to modify/return more than a single value.
- Library functions (e.g., `printf`) documented in `man` page. To use them, be sure to include the appropriate `#include`.

Slide 11

Sidebar — Compiler Options

- Earlier I showed the simplest way to use `gcc` to compile a program. But there are many variations — *options*. Specify on the command line, ahead of name of input file.
- Some of the most useful:
  - `-Wall` and `-pedantic` warn you about dangerous and non-standard things.
  - `-std=c99` allows you to use full C99.
  - `-o` allows you to name the output file (default `a.out`).

  (The right way to use all of these — makefiles, next time.)

Slide 12

Examples

- First let's write a program to calculate the roots of a quadratic equation, using the quadratic formula. (We'll hard-code input values for now — a discussion of getting input should wait until after we talk about pointers.)
- (To be continued.)