Slide 1

## Administrivia

- About the reading: You don't have to read every page carefully, but we won't have time in class to cover everything, so you should plan to at least skim.

Slide 2

## Getting Started with Linux (Review)

- (A UNIX person's response to claims that UNIX isn't user friendly: "Sure it is. It's just choosy about its friends.")
- The graphical system should give you a way to get a terminal window, which is what we will use a lot in this class (in keeping with the title!). In theory you know the basics from CSCI 1320. If you don't remember much, this might be a good time to review notes or whatever course materials you still have. (Or there should be something under "Useful links".)

Slide 3

## Useful Command-Line Tips

- The shell (the application that's processing what you type) keeps a history of commands you've recently typed. Up and down arrows let you cycle through this history and reuse commands.

  (Pedantic aside: "The shell" here means the one you're most likely to be using. There are other programs with similar functionality you could use instead.)

- The shell offers "tab completion" for filenames — if you type part of a filename and press the tab key, it will try to complete it.
- To learn more about command `foo`, type `man foo`. (This also works with C library routines — more about them later.) This is reference information rather than a tutorial, but usually very complete.

Slide 4

## Text Editors

- Many, many text editors, and people have favorites. I use and will teach in this class `vi`: It's found on every UNIX/Linux system I know of, and is very powerful, though it takes some getting used to. (`vi` on our Linux machines is actually `vim`, a more capable "clone" of the original `vi`.) Other popular Linux text editors include `emacs`, `pico`, and various graphical editors that come with "desktop environments" such as GNOME and KDE.
- Tip: If you're struggling with whatever editor you previously used, either spend a little time learning its features, or choose another one! `vim` has `vimtutor`. `emacs` also has built-in tutorial.

**Slide 5**

## 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 6**

## 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).
- Still others (including Scala and Python, sometimes) take an intermediate approach — initially compiled into *byte code* (object code for a made-up processor), which is (in principle) interpreted by a runtime system, with system library code brought in at runtime. (In practice, a "just-in-time" compiler may translate byte code into native object code on the fly.)

Slide 7

## 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.
- Syntax should look familiar to Java programmers (no accident — Java was designed that way). Less familiar to Python and Scala programmers.

Slide 8

## 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. (Some additions will work in `gcc` only with `-std=c99` option.)
- Still-later standards exist but are not (yet?) widely implemented.

Slide 9

## 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`.
- What does all this mean? first . . .

Slide 10

## A Few Words About Types

- To the hardware, "it's all ones and zeros"; types say how we want to interpret them (integers? characters?), define what kinds of things we can do with particular chunks of data.
- Should be reasonably familiar to Scala programmers but may be new to Python programmers. Both languages are more willing to guess your intent than C is. Book lists C's built-in types. Some will work in `gcc` only with `-std=c99`.

Slide 11

## Functions

- C programs are organized in terms of *functions* — a somewhat more primitive version of methods as found in object-oriented programming languages such as Python and Scala. As in other programming languages, C functions are a little like mathematical functions, except that evaluating them can have "side effects".

  (For example, evaluating the library function `printf` has the side effect of writing some text to standard output (by default, displaying it in the terminal window).)

- Unlike in some other languages, C functions have to be declared (or defined) before being referenced. Declaration includes name, return type, and formal parameters. For library functions, declaration is usually supplied via a `#include` preprocessor directive.

Slide 12

## Functions, Continued

- A complete C program must contain a function called `main`. It can be declared to take zero parameters, or two. Which to use? Depends on whether it needs access to command-line arguments. It should return an integer.
- When you execute a compiled/linked program, the operating system calls `main`, optionally passing it any command-line arguments. The program ends when this function does; its return value can be used to indicate whether the program succeeded (e.g., in shell scripts).
- (Now look again at our "hello world" program. More of it should make sense.)

Slide 13

## 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. `-Wall` *highly* recommended.
  - `-std=c99` allows you to use full C99.
  - `-o` allows you to name the output file (default `a.out`).
- Automate with `make` (more later).

Slide 14

## Variables in C

- To do anything interesting in a program, we need some place to store input and intermediate values — "variables".
- In C, variables must be *declared*, with a *name* and a *type*. In C89, all declarations must come before any code.
- Variable names follow rules for *identifiers* — letters, numbers, and underscores only, must start with letter or underscore, preferably letter. Case-sensitive.
- (To be continued . . . )

**Slide 15**

## Minute Essay

- Was anything today particularly unclear? Any other questions?