Slide 1

## Administrivia

- Your account for the Linux systems should have been set up Wednesday, and you should have received a password for it by e-mail.
- About minute essays, thanks for humoring my request to put “minute essay” and the course (1312 or ENGR) in the subject line.

  *Note* that I often don’t read these messages until the next day, so if you have a question that can’t wait, it’s better to send a separate message, but in any case put “URGENT” or the like in the subject line please.
- A request has been made to the TigerCard office to give you access to the classroom, so — soon?

Slide 2

## More Administrivia

- (From the slides for last time, somehow overlooked.)
- Be advised that this course, while it covers most of the same material as our first course for majors (CSCI 1320), doesn’t meet the prerequisite for the next course (CSCI 1321), so anyone seriously considering going further with CS should consider a last-minute switch of courses.

## Minute Essay From Last Lecture

Slide 3

- All ENGR majors, and everyone's had a fair amount of math.
- Many have used Matlab, and several have exposure to other programming-ilke tools and/or general-purpose programming language.
- Almost no one claims much experience with command-line environments and Linux. (So after a little more intro stuff we'll do that first.)

## Where To Start?

Slide 4

- In this course we have to do several things, and it's hard to know where to start:
- General discussion of programming and program design?
- Specifics of a programming language (C)?
- Specifics of tools (command-line environment, etc.)?
- I'll start with the first, briefly, then tools, then C?

Slide 5

## Solving Problems on Computers

- Appearances (maybe?) to the contrary, computers (considering only the hardware) are not smart. What they do well is perform sequences of simple math/logic operations very fast and very accurately.
- What makes them useful is that people have figured out how to break complicated tasks down into sequences of simple operations — i.e., how to "program" them.
- This requires a mindset not quite like that required for any other activity — and can involve a lot of creativity.
- It also involves a form of experimentation (which is why our introductory courses met the "Using Scientific Methods" requirement in the old Common Curriculum).

Slide 6

## Steps in Solving Problems on a Computer

- Understand the problem — what do you want the computer to do, exactly? (Not as simple as it might sound!)
- Design a solution suitable for a computer ("develop an algorithm").
- Implement the solution ("write the program"). Requires expressing your ideas in "a programming language" — and there are many! Programming languages similar to human languages in some ways, different in others; meant to be (somewhat!) human-readable while still being precise enough for a computer to understand.
- Test your solution. Involves the use of some tool that translates what you write ("source code") into something the computer hardware can work with.

Slide 7

## Solving Problems on a Computer, Continued

- Overall process — understand the problem, develop and test a solution — mostly independent of choice of programming language and platform (combination of hardware and operating system, roughly). So once you understand the principles, relatively easy to learn new languages.
- Opinions about which language to learn first, and on what platform, vary. For this course we will use C, mostly at the request of ENGR. Not as easy to use as some other choices, but one we and they could agree on. Not used a lot in general-purpose application development these days but still in use for "embedded systems" (which you might work with) and closer to the hardware. We will also do most work from the command line under Linux.

Slide 8

## Programming Basics

- What computers actually execute is *machine language* — binary numbers each representing one primitive operation. Once upon a time, people programmed by writing machine language (!).
- Obviously tedious and error-prone. Very early bright idea was to write something more human-readable (*source code*) and *have the computer translate it*. Useful even if the source code is just a human-readable version of the primitive operations (*assembler language*). Even better if the source code is less primitive (*high-level language*).
- Source code is simply plain text (as opposed to text plus formatting, as in a word-processor document). Since the hardware doesn't understand it, however, . . .

Slide 9

## Programming Basics, Continued

- Source code can be *interpreted* — translated line by line into something the hardware can understand, by another program called an *interpreter*.

  (This is how “scripting languages” work. An example is the command shell’s language. !)
- Or it can be *compiled* — translated by a program called a *compiler* into something the hardware can execute directly.

  (This is how traditional “high-level” languages such as C and Fortran work.)
- Or it can be compiled into some intermediate form that can be executed by another program.

  (This is how some recent languages such as Java and Scala work.)

Slide 10

## Writing Source Code

- How do you get source code? If using an interpreter, you can just type it in. If you want something you can keep and reuse, however, you need a tool that will do that.
- Simplest way to create source code is with a *text editor* — a program for writing and editing plain text. This is what we will do for now.
- (Another way is to use an *IDE* (Interactive Development Environment). We won’t use one of those in this class, but if you work on larger projects and/or in other programming languages you may want to.)

Slide 11

## A Word About Tools

- In this class we use Linux and command-line tools. We think it's important that CS majors know something about this environment, and we think it can be useful for people in other STEM disciplines as well — and we hope "new-to-me and different" has some appeal for all!
- (What is Linux? an operating system, as Windows and Mac OS X are operating systems; one of a family of operating systems descended from UNIX, developed at Bell Labs in the 1970s. A lot of servers run Linux or some other UNIX-like system; there are also many variants for the desktop.)
- A UNIX person's response to claims that UNIX isn't user-friendly: "Sure it is. It's just choosy about its friends."

Slide 12

## Getting Started with Linux

- When you log in, you should get a graphical desktop, which should be navigable with what you know from using other graphical environments (though some details are different).
- In Linux, we talk about files and directories; the idea is the same as Windows's files and folders, though again some details are different.
- The graphical system should give you a way to get a terminal window. Once you have that . . .

Slide 13

## Getting Started With the Command Line

- What you get when you start a terminal window is a “command shell”, similar to Windows’ “MS-DOS prompt”.

  Rather than pointing and clicking, you type the name of the program you want to run, plus whatever arguments (parameters) it needs.
- (Why would you want to use a command line? because for some things it’s arguably more efficient, and it’s “scriptable” in ways that GUIs typically aren’t.)
- Let’s try some commands . . . (Don’t worry if this goes by quickly — you should plan anyway to spend some time outside class trying out what we do in class and what’s in the “reading” (video lectures) for next time.)

Slide 14

## Some Commands

- `pwd` shows the current directory. (When you give a filename, it’s relative to this directory unless you give a full pathname.)
- `ls` lists the current directory. Add `-l` to get more information, `-A` to get “hidden” files too.
- `cd foo` changes to directory `foo`. Just `cd` goes back to your home directory. Try `cd Local` and then `ls`.
- `mkdir foo` creates a director `foo`. Might be useful to create one for your files for this class (call it csci1312, maybe, or CS1).
- `passwd` changes your password. (Not a command you’ll want often, but maybe now!) Note that this command does *not* echo what you type.

Slide 15

## Remote Access

- One of the strengths of a command-line enviroment is that it works well for “remote access” (using the computer when you aren’t sitting in front of it).
- To do this from another UNIX-like computer, use `ssh`. `scp` and `sftp` can be used to copy files.
- From a Windows computer, install either Cygwin (UNIX-like toolkit) or PuTTY (terminal emulator). I’ll send e-mail with details, but be advised that one option — which will work even from off campus — is to connect to Trinity’s VDI system, which has PuTTY installed.

Slide 16

## Minute Essay

- Anything today that was particularly unclear?