Slide 1

Administrivia

- (Reading assignment for today updated/expanded. Read for next time!)

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.")
- 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).
- 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 not, review the relevant chapter of the book Dr. Lewis is writing for POP I/II.

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

C Versus Python/Scala — Recap

- Language-level differences: C is pretty low-level compared to both Python and Scala. Programmer must do more things explicitly, and language/environment has fewer safety rails. Probably not a great choice for writing applications, but good for low-level work (such as o/s-level programs) and educational.
- Development environment differences: No REPL; code must be compiled into "executable", which then runs like other executables. Compiler error messages not always novice-friendly.

Slide 6

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 7

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.

Slide 8

A First C Program, Revisited

- Last time we wrote the traditional "hello world" program in C, compiled/linked it (with `gcc`) and executed it.
- Now let's look at the program line by line. But first . . .

Slide 9

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 the `-std=c99` option.

Slide 10

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 11

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 12

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 13

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.

Slide 14

Types in C

- Integer types include `int`, `short`, `long`. (All can be declared `unsigned` too.) Unlike in some language (such as Java), sizes not 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.)
- Floating-point types include `float`, `double`. Binary equivalent of scientific notation (with exponent and mantissa). Minimum size for `double` is larger than for `float` so allows more significant figures, larger range.
- More about other types later.

Expressions in C

- C (like many other programming languages) has a notion of an *expression*. Simple examples (assuming we've declared variables `x` and `y`):
  - `5`
  - `x`
  - `y + 5`
  - `(x + y) / 2`
- Every expression has a *value*, and computing this value is called *evaluating the expression*. Evaluate the above expressions, assuming `x` has value 10 and `y` has value 20 . . .

Expressions in C, Continued

- Sometimes evaluating an expression also produces changes to variables in the expression or other variables; these are called *side effects*. Examples:
  - `x = 10`
  - `printf("hello, world\n)`

  (Yes, really! Usually we don't care about much about the values of these expressions, just their side effects.)
- Many, many operators of different kinds. For now we'll look only at the ones for arithmetic.

Slide 17

Arithmetic Expressions — Operators

- Usual arithmetic operators `+`, `-`, `*` (multiplication), `/` (division). (`+` and `-` can be unary too.)

  Notice that division, applied to integers, discards any remainder. This is so the result will be an integer too, and can even be useful. What if you want a fraction? Later.

- Also `%` operator for getting remainder; e.g., `x % 2` is 0 if `x` is even, 1 if it's odd.
- Other useful arithmetic operators include pre/post increment/decrement, bit shifts.
- Expressions can be quite complex. How they're evaluated depends on rules of precedence and associativity. My advice — when in doubt, use parentheses! Example: `(x + y) / 2` versus `x + y / 2`.

Slide 18

Statements in C

- C programs are made up of *statements* (usually collected inside *functions*.
- Statements come in several types:
  - Null (`;`).
  - Expression (*expression* `;`).
  - Return (`return` *expression* `;`).
  - Compound (more later).

Output

Slide 19

- The “hello world” used `printf` to print some text. `printf` can do a lot more.
- For example, we can use it to print integers, e.g.,

```
printf("the value of x is %d\n", x);
```

Sidebar — Man Pages, Revisited

Slide 20

- As mentioned earlier, most commands — and many library functions — have “man pages” (short for “manual”). These are meant as online references rather than tutorials, so not always easy reading, but usually very complete.
- `man` program shows its output to you using a program intended for paging through text. On our systems, default is `less`. Keystroke commands include space to go forward, `b` to go back, `q` to quit. `h` for help — or, of course, you could read all about it (how?).
- Sometimes there are multiple commands/functions with the same name. `printf` is one. `man printf` tells you about the (command-line) command, not the C library function. To get all possibilities, `man -a printf`. To get the one for the library function, `man 3 printf`.

Minute Essay

- Was anything today particularly unclear? Any other questions?