

Slide 1

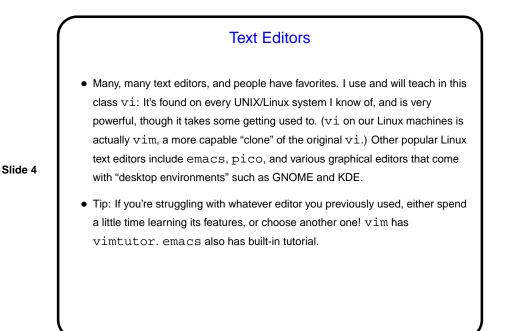
# Getting Started with Linux (Review) (A UNIX person's response to claims that UNIX isn't user friendly: "Sure it is. It's just choosy about its friends.") The graphical system should give you a way to get a terminal window, which is what we will use a lot in this class (in keeping with the title!). 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".)

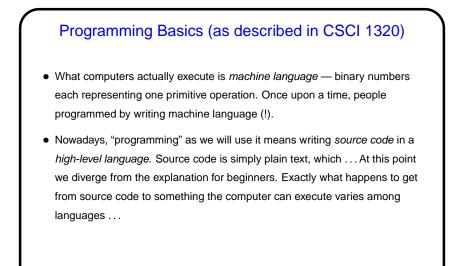
### **Useful Command-Line Tips**

• 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.

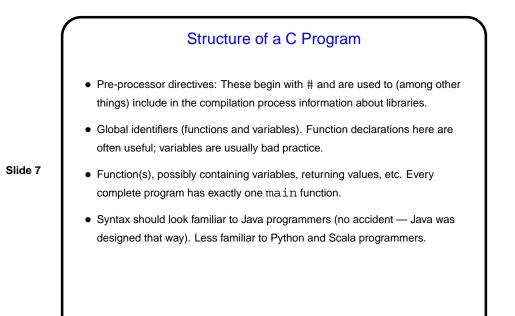




Slide 5

# From Source Code to — What?

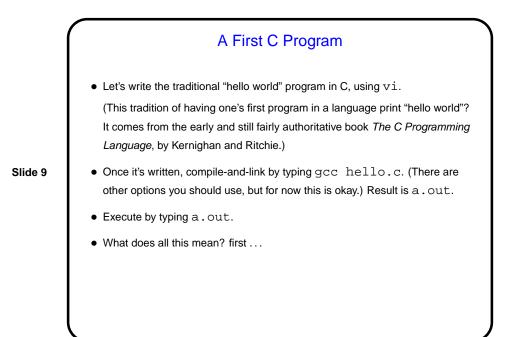
- Some high-level languages (such as the language understood by typical UNIX/Linux command shells) are directly interpreted by some other program.
- Others are *compiled* into *object code* (machine language) and then *linked* with other object code (including system libraries) to form an *executable* (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.)

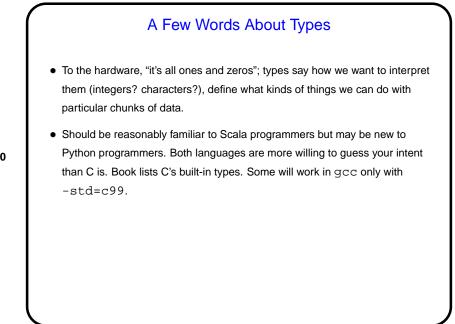


A Few Words About "Old C" Versus "New C"

- 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.







# **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.

### 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 11

