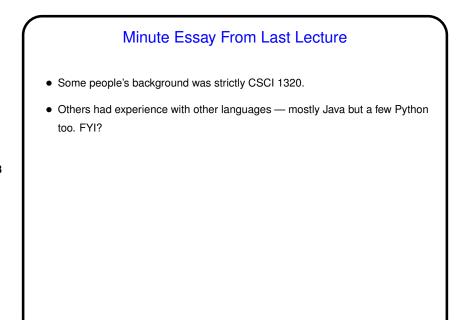


More Administrivia
Homework 2 on the Web; due next week. Not algorithmically challenging, but the first program in a new language doesn't need to be?
Most (non-trivial) code from class will be on the course Web site, under "sample programs".
You can do homeworks with just a text-mode connection (e.g., via PuTTY), but then turning in the files is a pain. But you can use the "mail files" script on the "sample programs" page (instructions there too).
Our student ACM chapter will be doing peer tutoring for intro courses, including this one, 5pm–9pm M/T/W/R in this room. Details in e-mail. Excellent resource for getting help with homeworks!



Slide 3

Programming Basics (as described in CS1)

• What computers actually execute is *machine language* — binary numbers each representing one primitive operation. Once upon a time, people programmed by writing machine language (!).

Slide 4

• 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 ...

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).

Slide 5

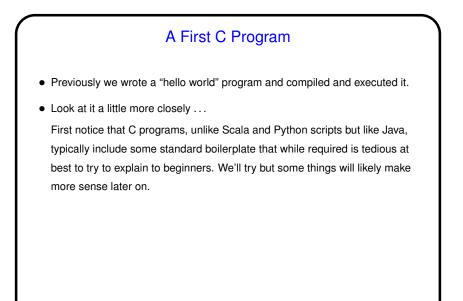
Still others (including Scala, Java, and sometimes Python) take an
intermediate approach — initially compiled into *byte code* (object code for a
made-up processor), which is (in principle) interpreted by a runtime system,
with system library code brought in at runtime. (In practice, a "just-in-time"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

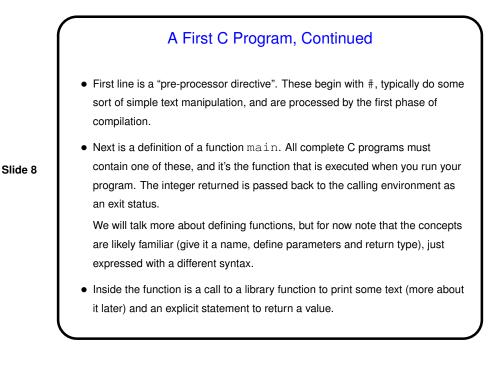
A Little About C

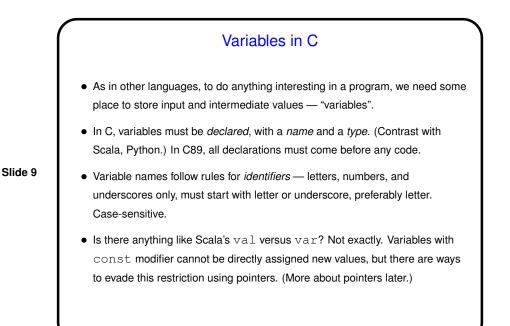
 Many languages you encounter nowadays were designed to provide a relatively-easy-to-use environment that's the same from platform from platform. Full implementation may require a fairly featureful platform (e.g., ability to support graphics).

Slide 6

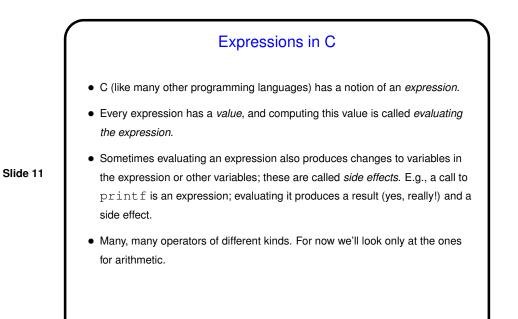
 C, in contrast, was designed to be easier to use and more portable than assembly language, while still being implementable on a very wide range of platforms in a way that produces programs that are as efficient as is reasonably possible. The language standard(s) defines things all implementations must do while leaving some details (e.g., sizes of various numeric types) up to the implementer. Note that there have been multiple official standards (C89, C99, etc.), and later standards include more features.







Slide 10
Integer types include int, short, long. (All can be declared unsigned too.) Unlike in some language (such as Java and Scala), 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.



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.

