Slide 1

Administrivia

- Reminder: Homework 1 due Friday at 11:59pm. Several people have turned it in, and skimming what's been turned it I gather that many people learned helpful things. Some suggestions:
- Editor suggestions:
  - Consider making yourself a "cheat sheet" (paper or electronic) of commands you think you will find most useful, to have readily available when using this tool, so you're less tempted to do things the hard way ("insert" mode, backspace, arrow keys, and not much else).
  - If you didn't find out things you wanted to do (cut/copy/paste lines, e.g.), consider going through more of the tutorial.

Slide 2

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

Minute Essay From Last Lecture

- Some people's background was strictly CSCI 1320.
- Others had experience with other languages — mostly Java but a few Python too. FYI?

Slide 4

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 (!).
- 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 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, 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.)

Slide 6

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).
- 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 7

A First C Program

- Previously we wrote a "hello world" program and compiled and executed it.
- Look at it a little more closely . . .

  First notice that C programs, unlike Scala and Python scripts but like Java, typically include some standard boilerplate that while required is tedious at best to try to explain to beginners. We'll try but some things will likely make more sense later on.

Slide 8

A First C Program, Continued

- First line is a "pre-processor directive". These begin with `#`, typically do some sort of simple text manipulation, and are processed by the first phase of compilation.
- Next is a definition of a function `main`. All complete C programs must contain one of these, and it's the function that is executed when you run your program. The integer returned is passed back to the calling environment as an exit status.

  We will talk more about defining functions, but for now note that the concepts are likely familiar (give it a name, define parameters and return type), just expressed with a different syntax.
- Inside the function is a call to a library function to print some text (more about it later) and an explicit statement to return a value.

Slide 9

Variables in C

- As in other languages, 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*. (Contrast with Scala, Python.) 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.
- Is there anything like Scala's `val` versus `var`? Not exactly. Variables with `const` modifier cannot be directly assigned new values, but there are ways to evade this restriction using pointers. (More about pointers later.)

Slide 10

Types in C

- 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.

Slide 11

Expressions in C

- C (like many other programming languages) has a notion of an *expression*.
- Every expression has a *value*, and computing this value is called *evaluating the expression*.
- Sometimes evaluating an expression also produces changes to variables in the expression or other variables; these are called *side effects*. E.g., a call to `printf` is an expression; evaluating it produces a result (yes, really!) and a side effect.
- Many, many operators of different kinds. For now we'll look only at the ones for arithmetic.

Slide 12

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.

Slide 13

Pre/Post Increment/Decrement

- (These four operators are likely new to Scala programmers.)
- `x++` and `++x` both have the side effect of adding 1 to `x`, but considered as expressions they have different values (before-increment and after-increment respectively). Similarly for `x--` and `--x`.
- Often used solely for side effect (e.g., as a substitute for the more-verbose `x+=1`), but not always (i.e., sometimes used in contexts where expression value matters too).

Slide 14

Simple Output

- Simple/typical way to produce output (to "standard output" — terminal for now) is with library function `printf`.
- Parameters are "format string", which may include "conversion specifications", followed by zero or more expressions, one for each conversion specification. E.g., to print value of `int` variable `x`:

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

Full details in `man` page for `printf`. (Find with `man 3 printf`.)

Slide 15

Simple Input

- Simple way to get integer/float input (from "standard input") is with library function `scanf`. Parameters are "format string" (similar to the one for `printf`) and list of pointers (more later) to variables, e.g.:

  ```
  scanf("%d %d", &var1, &var2);
  ```

  Behaves somewhat like library functions for reading from standard input in other languages, except that it skips whitespace (including newlines) and stops when it encounters something other than what it needs (e.g., non-numeric characters when number is wanted).

- Considered as an expression, call to `scanf` has a value, namely the number of variables successfully read. C-idiomatic way to check for success is

  ```
  if (scanf("%d %d"&var1, &var2) == 2) ....
  ```

  (More when we talk about conditionals next time.)

Slide 16

Compiling with `gcc` — A Tip

- With the extra flag `-Wall`, `gcc` will potentially print (more) warnings about code that is legal but may not work as intended.
- These messages can be very helpful! so I recommend that you *always always* use it — e.g.,

  ```
  gcc -Wall hello.c
  ```

Slide 17

Minute Essay

- What are you planning to do about the textbook? if you want to buy hardcopy from the bookstore — do they have them in yet?
- Any questions?