Slide 1

Administrivia

- None.

Slide 2

Minute Essay From Last Lecture

- Why is the `printf` so different from `println`?

  Partly I'd say it's more low-level — language does less for you — but also it gives more control. Notice that Scala has a `printf`!

- How do data types in C compare to those in Scala?

  Simple ones are very similar *except* that in C there's some flexibility about precision (more in another slide).

Slide 3

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 4

Variables — Review/Recap

- In order to do anything useful we usually (though not always!) need some variables. In C, variables must be *declared* before being used. (Contrast with Python.) Declaration specifies name and type. (Contrast with Scala.)
- Once you have variables, you can assign values to them, using *expressions* that range from simple constants to complex math-like formulas involving constants and/or other variables.

Slide 5

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 . . .

Slide 6

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 7

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 8

Expressions — "Caveat Programmer"

- C standard is somewhat imprecise about details of expression evaluation — e.g., in evaluating

  ```
  f() + g()
  ```

  two functions could be called in either order. (Why? To allow greater flexibility for implementers, possible allow for more-efficient programs.)
- C syntax allows programmers to write statements/expressions in which a variable's value is changed more than once, e.g.,

  ```
  i = (i++) + (i--);
  ```

  Syntactically legal, but standard says that such expressions invoke "undefined behavior". Best to avoid that!

Slide 9

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).

Slide 10

Output

- 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);
```

Slide 11

Preprocessor Directives — A Bit More

- Examples so far have started with `#include` directive to tell compiler where to find information about I/O library functions. (Roughly — "include", i.e., copy, information from header file-or-equivalent.) This is input to the "preprocessor".
- Another useful directive is `#define`, to give meaningful names to constants, e.g.,

```
#define IMPRECISE_PI 3.14159
```

Slide 12

A Few Words About Syntax

- Python programmers should note that in C, unlike in Python, indentation is not generally syntactically significant. (But adopting a consistent style makes your code more readable to humans.)
- Scala programmers should note that in C, unlike in Scala, the compiler will not add semicolons to the ends of statements or guess about types.

Slide 13

Simple Output, Revisited

- 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.

Slide 14

Simple Input

- Simple way to get integer/float input (from "standard input" — keyboard for now) is with library function `scanf`. For now we will look only at simple forms:

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

  etc. Parameters similar to `printf`, except for that ampersand. (It generates a pointer. More about that later!)
- Considered as an expression, call to `scanf` has a value, namely the number of variables successfully read. (So you can check it to make sure valid input was entered.)

Slide 15

Minute Essay

- None — sign in.