Slide 1

Administrivia

- Homework 1 was due today. Deadline extended to next Wednesday.

Slide 2

C Basics — Quick Overview

- Unlike Python and Scala scripts (but like Java programs), C programs include some standard boilerplate (`#include` for library functions, explicit `main`).
- Variables must be explicitly declared, including type.
- Expressions similar to those in Python/Scala/Java but with a few differences.
- Statements are also similar, but assignments are considered to be expressions too, with a value. Allows chaining, e.g.,

  ```
  a = b = 10;
  ```

- A caveat: The C standard does not spell out everything (e.g., size of `int` type) so experimental results are not necessarily conclusive (might be specific to a particular compiler/system).

Slide 3

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

- First ANSI standard for C — 1989 ("C89"). Widely adopted, but has some annoying limitations.
- Later standard — 1999 ("C99"). 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 standard (2011) exists but is not (yet?) widely implemented.

Slide 4

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*. (Contrast with Python, Scala.) 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 5

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 6

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 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.

Slide 8

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 9

Expressions — "Caveat Programmer"

- 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`.
- 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 10

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 11

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) ....
  ```

Slide 12

Sidebar — Man Pages, Revisited

- 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`.

Slide 13

Minute Essay

- A student sent me the following program and asked why it printed zero for both `x` and `x+1`. What's going wrong? (Hint: `gcc -Wall` gives warnings.)

```
#include <stdio.h>
int main(void) {
    int x = 0;
    printf("hello world\n");
    printf("x = %.2f x+1 = %.2f\n", x, x+1);
    return 0;
}
```

Slide 14

Minute Essay Answer

- `gcc` prints these messages:

```
qq.c:7: warning: format %.2f expects type double, but argument 2 has type int
qq.c:7: warning: format %.2f expects type double, but argument 3 has type int
```

  which should tip you off to the mismatch between the type of the expressions (`int`) and the format.