Slide 1

Administrivia

- A request: You will turn in most if not all work for this course by e-mail. Please include the name or number of the course in the subject line of your message, plus something about which assignment it is, to help me get it into the correct folder for grading.

Slide 2

Programming Basics (as described in CSCI 1320)

- 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 3

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.)

Slide 4

Why Learn C? (For Java/Python/Scala Programmers)

- Java (and Scala and Python) provides a programming environment that's nice in many ways — lots of safety checks, nice features, extensive standard library. But it hides a lot about how hardware actually works.
- C, in contrast, has been called "high-level assembly language" — so it seems primitive in some ways compared to many other languages. What you get (we think!) in return for the annoyances is more understanding of hardware — and if you do low-level work (e.g., operating systems, embedded systems), it may well be in C.

Slide 5

Structure of a C Program

- Pre-processor directives: These begin with # and are used to (among other things) include in the compilation process information about libraries.
- Global identifiers (functions and variables). Function declarations here are often useful; variables are usually bad practice.
- Function(s), possibly containing variables, returning values, etc. Every complete program has exactly one `main` function.
- Syntax should look familiar to Java programmers (no accident — Java was designed that way). Less familiar to Python and Scala programmers.

Slide 6

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. What we do in this class will focus on older standard for this reason.

Slide 7

A First C Program

- Let's write the traditional "hello world" program in C, using `vi`.

  (This tradition of having one's first program in a language print "hello world"? It comes from the early and still fairly authoritative book *The C Programming Language*, by Kernighan and Ritchie.)
- Once it's written, compile-and-link by typing `gcc hello.c`. (There are other options you should use, but for now this is okay.) Result is `a.out`.
- Execute by typing `a.out`.
- Now let's look at the program line by line . . .

Slide 8

Functions

- C programs are organized in terms of *functions* — as in other programming languages, they're 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).
- A complete C program must contain a function called `main`. When you type `a.out`, the operating system calls this function. The return value can be used to indicate whether the program succeeded.

Slide 9

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*. In C89, declarations must come before code.
- Variable names follow rules for *identifiers* — letters, numbers, and underscores only, must start with letter or underscore, preferably letter. Case-sensitive.
- Variable types? To the computer, "it's all ones and zeros"; types say how we want to interpret them (integers? characters?), define what kinds of things we can do with a variable. Tutorial lists C's built-in types. Some will work in `gcc` only with the `-std=c99` option.

Slide 10

Variable Types in C

- Integer types include `int`, `short`, `long`. (All can be declared `unsigned` too.) Unlike in some language (such as Java), sizes of 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.

Slide 11

Variable Types in C, Continued

- No Boolean type in C89, so programmers often use integers.
- `char` is an ASCII (not Unicode) character.
- Arrays represent collections of identical items (more about them later).
- Pointers provide a way to indirectly reference variables (more about them later).

Slide 12

Sidebar — Compiler Options

- Earlier I showed the simplest way to use `gcc` to compile a program. But there are many variations — *options*. Specify on the command line, ahead of name of input file.
- Some of the most useful:
  - `-Wall` and `-pedantic` warn you about dangerous and non-standard things. `-Wall` *highly* recommended.
  - `-std=c99` allows you to use full C99.
  - `-o` allows you to name the output file (default `a.out`).
- Automate with `make` (more later).

Slide 13

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 14

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 15

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 16

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 17

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 18

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 19

Functions

- Functions similar in concept to those in many other programming languages, with a couple of key distinctions:
  - They have to be declared (or defined) before being referenced. Declaration includes name, return type, and formal parameters.
  - Pass-by-value semantics for parameters means you need pointers if you want to modify/return more than a single value.
- Library functions (e.g., `printf`) documented in `man` page. To use them, be sure to include the appropriate `#include`.

Slide 20

Minute Essay

- Was everything clear today?
- Is the reading making sense to you?