

Administrivia

Slide 1

- As noted in e-mail, I put a link in TLEARN to the course Web page, so you can find it that way if that's easier to remember.
- "Useful links" on course Web site has links to information about UNIX/Linux commands, etc.
- For minute essays, put "minute essay" *and the course name or number* in the subject line. (Most class days I teach multiple courses, so this helps me quickly and reliably pick out the minute essays for each one.)
You can ask me anything course-related, but if your question needs a quick reply, *please* put "urgent" in the subject line.

More Administrivia

Slide 2

- Reminder: Homework 1 due Friday (by midnight).
- Homework 2 on the Web; due next week. Not algorithmically challenging, but the first program in a new language doesn't need to be?
Note request to tell me about collaboration. I will fill in details probably tomorrow and send mail.
- About the reading: You don't have to read every page carefully, but we won't have time in class to cover everything, so you should plan to at least skim.
- Most (non-trivial) code from class will be on the Web, under "Sample programs".

Slide 3

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 4

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 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.)

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 5

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 notice 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 6

Variables in C

Slide 7

- 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.)

Types in C

Slide 8

- 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.

Expressions in C

Slide 9

- 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.

Arithmetic Expressions — Operators

Slide 10

- 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.

Pre/Post Increment/Decrement

Slide 11

- (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).

Simple Output

Slide 12

- 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`.)

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 13

Minute Essay

- Any questions?

Slide 14