Slide 1

Administrivia

- Slides from class will be on Web — preliminary version shortly before class, final version later that day.
- Code shown in class will usually be on the Web, linked from the "Sample programs" page here. (This doesn't matter much now, but if you want the "hello world" program, it's there.)
- Reminder: *Please* do not reboot the machines in this room (HAS 340); people rely on their being available for remote access.

  Also be careful not to inadvertently shut them down when trying to log off.

  If a previous user has left the machine's screen locked, you can use use control-alt-backspace to restart the graphical subsystem.

Slide 2

"What is a Computer" — Hardware Components

- Input and output devices — connect computer with outside world.
- CPU(s) — what actually does the work (the "brain"). Executes a sequence of very primitive instructions, such as "fetch a number from memory" or "add two numbers". Details of these instructions differ among machines.
- Main memory — very long list of addressable locations, something like a whiteboard marked off into cells you can write into, read from.

  (Ultimately, everything represented in the form of *bits* — ones and zeros, off/on switches. Interpret these as numbers, text, . . .)
- Secondary memory — more permanent but less accessible version of main memory, these days usually disk.

Slide 3

“What is a Computer” — Software

- For practical purposes, computer is always executing some program.
- During execution, programs are stored in main memory (encoded in a way that corresponds to the primitive operations of this particular computer). Part of the computer’s state is address of next instruction to be executed.
- Operating system — the program that runs “all the time”. Examples: Windows, MacOS, DOS, OS/2, UNIX, VMS, MVS, etc., etc. For the most part, it’s talked to by other programs.
- Application programs — what users normally interact with. Examples: word-processing programs, spreadsheets, Web browsers, games, etc.

Slide 4

Where Do Programs Come From?

- In early days, programmers wrote in the machine’s language – the primitive instructions the machine understands. Can still do that, but a lot of work, and only works for one kind of machine.
- Another way is to write in *high-level language* — more abstract, less detailed, somewhat closer to how humans think — and have a program that translates this into the machine’s language.
- We call the input to these translation programs *source code* their output *object code*, and the programs *assemblers* or *compilers*.

  (Where does source code come from? It’s plain text, and can be produced using a text editor such as `vi`, or with other tools — but typically *not* with a word processor. (Look at a Word file with `vi` sometime!))

Slide 5

Where Do Programs Come From?, Continued

- Is object code a complete program ready to be executed? Maybe. But typically some common operations (I/O, e.g.) are provided via a *library* (of object code), and to get complete program you combine your object code with library via a *linker* to get an *executable*. For simple programs, compiling and linking often combined into one step.
- This executable (file) can be loaded into memory by the operating system and executed. This is where most applications come from — `ls`, `vi`.

  (Compare results of `file a.out` (that we produced) to result of `file /bin/ls`.)

Slide 6

Recap — Linux Command-Line Environment

- Notion of "home directory".
- Commands to work with directories (folders): `cd`, `mkdir`, `rmdir`, `pwd`, `ls`.
- Commands to work with files: `cp`, `mv`, `rm`, `vi`.
- Commands to get information: `man`, `man -k` (a.k.a. `apropos`). Works for most commands and some functions (more about that later).

  Read man page for `less` to understand how to page through information.

  `man -a foo` gives all man pages for `foo`. Example: `printf`.

Slide 7

File Permissions in UNIX/Linux

- Access to files specified in terms of three categories of users (owner, group, and other) and three kinds of access (read, write, and execute).
- To show permissions, `ls -l`. First character says directory/not, then three groups of three letters each (rwx), one for each category of user. Example:

  ```
  -rw------- 1 bmassing bmassing 115 2007-08-30 10:07 hello.c
  ```

- To change permissions, `chmod`. Can specify via octal (base 8) numbers, but usually easier to use symbolic mode. Examples:

  `chmod go= foo` to say only owner can access `foo`.

  `chmod go+r foo` to say everyone can read `foo` (but not necessarily write it).

Slide 8

Remote Access to the Lab Machines

- From another UNIX/Linux machine: Open a terminal window, type `ssh user@machine`.

  (This includes Mac OS X.)

- From a Windows machine, use PuTTY or Cygwin. Also should be able to access home directory on Sol (department Linux file server) from Windows. (Quick demo, time permitting.)
- Also see my Web page on remote access here. It has links to where you can download PuTTY and Cygwin.

  Access from off-campus is possible but requires that you tell us (the CS admin people) your IP address. Come talk to me, or send mail.

Slide 9

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. More about all of this later.

Slide 10

Comments

- Anywhere in the program you can include *comments*, meant for human readers and ignored by the compiler. The old C style is to start with `/*` and end with `*/`. The book shows a newer style.
- I will nag you a lot about putting in good comments: They can be very useful for human readers (me now, you if you look at your programs again in a year, someone else if you program as part of a team, etc.).

  Also can be a good way to “think out loud” about a program before starting to code.

Slide 11

Minute Essay

- Most general-purpose computers you've seen these days have CPUs that execute the "x86" instruction set. But some don't (e.g., IBM mainframes). Do you think an executable (file) produced by `gcc` can be run on an IBM mainframe? If not, why?
- What if we took the source code and compiled it on the IBM? Would it compile? Would the resulting executable be the same as compiling it on one of our lab machines?

Slide 12

Minute Essay Answer

- No, generally speaking the executable produced on one kind of machine won't run on another — the low-level operations of the processor and the way they're encoded might be different.
- The same source code can be used to produce executables for different kinds of machines. (That's one of the benefits of high-level languages.) On each machine, you use a compiler tailored for to that type of machine, and it produces an appropriate executable.