CSE 303:
Concepts and Tools for Software Development

Hal Perkins

Spring 2008

Lecture 1— Course Introduction

Welcome!

We have 10 weeks to move to a level well above novice programmer:

- Command-line tools/scripts to automate tasks
- C programming (lower level than Java; higher than assembly)
- Tools for programming
- Basic software-engineering concepts
- Basics of concurrency
- Societal/ethical implications of computing

That's a lot!

Get used to *exposure*, not *exhaustive investigation*.

Today

Today in class:

- Course mechanics
- Course overview and motivation
- Dive into the *command shell*

By next time:

- Do homework 0 (worth 1 point!) get it done?)
- Work through the "getting started guide" including changing-your-shell.

Who and What

- 3 class meetings (slides, code, demos, questions)
  - Material on-line (often afterwards), but take notes
  - Advice: jot down keywords so you can better look stuff up later
  - Advice: use class for concepts (a debugger allows you to interrupt programs and inspect values) and documentation for details (b is gnu-debugger abbrevation for breakpoint). (Class will do more "organizing" than "teaching".)
  - Advice: Try stuff out the same day. Experiment!!
  - Warning: The slides are *NOT nearly enough* for learning the material. They are an outline only.
- Office hours (Kari, Dingding, Pamela, me)
  - Advice: use them

Homework and Exams and tentative grading

- 7 homeworks (+/- 1) (45%)
  - 2 on shells and shell scripting
  - 2-3 on C
  - 2-3 on programming tools and methodologies (1 in small teams)
- 1 *short* paper on societal implications (10%)
  - More on this later
- 1 midterm (20%) and 1 final (25%)

Collaboration: Mostly individual work; *never* look at or show homework code to others.

Extra Credit: When available, small effect on your grade if you do it

Academic Integrity

Read every word of the course policy very carefully.

Always explain any unconventional action on your part.

Promoting and enforcing academic integrity has been a personal focus for many years:

- I trust you completely
- I have no sympathy for trust violations, nor should you

Honest work is the most important feature of a university. It shows respect for your colleagues *and yourself*.

Particularly fine line: Looking at similar shell scripts is useful!

What is this "303" thing?

303 is a relatively new course (first offered Spring 03)

A noticeable "laundry list of everything else" feel/place in the curriculum.

But there's a real common thread worth remembering:

There is an amorphous set of things computer scientists know about and novice programmers don't. Knowing them empowers you in computing, lessens the "friction" of learning in other classes, and makes you a mature programmer.

You "toss things in your mental purse" your whole career; 303 gives you a sense of what's out there and starts you on the path.

6 general areas

1. The command-line
   - Text-based manipulation of your computing environment
   - Automating (scripting) the manipulation
   - Using powerful *utility* programs

Quick-and-dirty ways to let the computer do what it's good at so you don't have to!

We will use Linux (an operating system) and bash (a *shell*), though it's irrelevant for the concepts.

Half the battle: Knowing the name of what "really ought to exist"

Half the battle: Programming in a language designed for interaction

6 general areas

2. C (and a little C++)

   - "The" programming languge for operating systems, networking code, embedded devices, ...
   - Manual resource management
   - Trust the programmer; a "correct" C implementation can run a program with an array-bounds error and *set the computer on fire*
   - A "lower level" view of programming where it can help to know that all code and data sits together in "one big array of bits".

Half the battle: Parts look like Java, but that can deceive you

Half the battle: Learning to think before you write, and test often

6 general areas

3. Programming tools

So far you have written programs and run them. There are programs for programming you should know about:

- Compilers (vs. interpreters)
- Debuggers
- Profilers
- Linkers
- Recompilation managers
- Version-control systems
- ...

6 general areas

4. Software-development concepts:

Stuff you may not need for 1e2 line programs, but how about 1e6?

- Testing methodologies
- Team-programming concepts
- Software specifications
- ...

6 general areas

5. Basics of concurrency

Programs where "more than one thing can happen at once"

- Brand-new kinds of bugs (e.g., races)
- Approaches to *synchronization*
- Increasingly important (lab machines have two processors — and so will your next laptop — if it doesn't have more!)

6 general areas

6. Societal/ethical implications of computing:

Being a professional/scientist/engineer requires confronting societal considerations.

We won't "teach politics" but we will think critically about computing issues challenging humanity because we cannot only leave it to politicians, lawyers, philosophers, ...

Examples: software patents, digital privacy, digital rights management, software licensing, software-engineer certification, the digital divide, accessibility, software security, electronic voting

View of a large world

1. The command-line
2. C
3. Programming tools
4. Software-development concepts
5. Basics of concurrency
6. Societal/ethical implications of computing

"There is more to programming than Java methods"
"There is more to software development than programming"
"There is more to computer science than software development"
"There is more to computing's effects than computer science"

So let's get started...

The O/S, the filesystem, the shell

Some things you might have a sense of but never were told precisely (may as well start at the beginning)...

- The file-system is a tree
  - (Actually it's a dag)
  - The top is /
  - Interior nodes are directories (displayed as folders in GUIs)
- Users *log-in*, which for Linux means getting a *shell*
  - They have *permissions* to access certain files/directories
  - They have a "home directory" somewhere in the file-system
  - They can run programs. A running program is a *process*. (Actually could be more than 1.)

File Access

You may be used to manipulating files via a GUI using WIMP.

You can do all the same things by running programs in the shell.

Just like an "explorer window", the shell has a *current working directory*.

It really helps to remember the names of key commands: `ls`, `cp`, `mv`, `rm`, `cat`, `cd`, `pwd`. (Most are really just programs.)

Current directory: `.`
Parent directory: `..`

Relative vs. absolute pathnames

Why would anyone want to interact like this?

- Old people who remember life before GUIs :-)
- *Power users* who can *go faster*
- Users who want easy logging
- Users who want easy instructions
- Users who want *programmability*

The last one will be the core of homeworks 1 and 2.

Most computer scientists use GUIs and shells, depending what they're doing.
Linux has GUIs and Windows has shells.

Options and `man` (and `info`)

Bad news at first: Program names and options are short, arcane, and numerous.

Good news:

- Most programs will print a *usage* argument if given bad options (or often `-help` or `--help`).
- The program `man` takes a program name and prints a file descrbing the program.
- The program `info` is a textual browser for some programs, particularly complex ones (`bash`, `gcc`, others)
- There are tons of other resources (e.g., the web).
- Decades of existence has led to standardized things:
  - Dashes for options, followed as necessary by option argument

More programs and options

- `less` (is `more`)
  - used by `man`
  - spacebar, b, /search-exp, q
- `chmod`
- `mail`

And some that aren't technically programs (more on this later)

- `exit`
- `echo`
- (`cd`)

The shell, again

The shell is an *interpreter* for a strange programming language (of the same name). So far:

- "Shell programs" are program names and arguments
- The interpreter runs the program (passing it the arguments), prints any output, and prints another prompt. The program can affect the file-system, send mail, open windows, etc.
- "Builtins" such as `exit` give directions to the interpreter.

It's actually much more complicated:

- (two kinds of) variables.
- some programming constructs (conditionals, loops, etc.)
- The shell interprets lots of funny characters differently, rather than pass them as options to programs.