CSE 490c (CSE 303)

Concepts and Tools for Software Development

Goal: novice programmer $\Rightarrow$ skilled developer

- Understand and control (and take responsibility for)
  - operating system
  - development environment
  - program's resources
- Can work in teams
- Know testing, design strategies
- Aware of impacts on society
- Self-reliant

Warning!

- Course is under construction
- Things will go wrong
- Feedback desired
- I'm teaching what I know and use
  - There's good stuff I don't know
  - You may know things I don't
  - Contribute!
- Take charge of getting the most out of this course

Tentative outline (part 1)

- Unix, advanced cmds, scripts: 1 week
- Dev. tools, group programming: 1 week
- Testing, specifications: 1 week
- C/C++: 2 weeks
- (midterm)

Tentative outline (part 2)

- Other dev. environments: 1 week
- Design patterns: 1 week
- Requirements, UI design: 1 week
  - (final)
Homework and projects

- Roughly weekly
- Some exercises
- An extended group project
  - A Java component
  - A C component
  - Testing & design & documentation parts

"Section"

- Friday will often be different
  - Discussion & debate on societal impacts of computer systems
  - Contribute your dippings
  - Group project work & meetings
  - Code reviews
  - Guest lectures
  - Contribute your knowledge

Textbooks

- C++ for Java Programmers
- Design Patterns
- (one other on being a good programmer)

Survey on background

- Unix?
- Java?
- C/C++?
- Program size?
- Development environments & tools?
- Can bring laptops?

Why study Unix?

- Contrast with insulating point-and-click OSs, like Windows
  - Understand and manage your own environment
- See a different kind of programming than Java or C programming
- See how simple (and sophisticated) tools can be combined to get interesting effects
  - pipes
  - scripts
### Unix is widely available
- Machines running Linux (and other Unix variants)
  - ceylon, fiji, sumatra, tahiti
- From Windows machines:
  - Can remotely log in to instructional Linux machines, e.g. using Ssh Secure Shell
  - Can install Cygwin
  - (How can you find out about Cygwin?)

### The shell
- When log in, get a command shell or interpreter
  - Can enter commands, see them execute
  - Line-oriented
- Standard syntax:
  - `commandName arg1 ... argN`
  - some args are options conventionally prefixed by a hyphen
- Start in *home directory*

### Directories and files
- A directory is a folder containing files and other (nested) directories
  - Directories form a tree
  - A directory is just a special kind of file
- Files (including directories) can have names of any length, including just about any characters, any number of times
  - No rules about 3-character extensions
  - hidden-blobbedFile

### Path names
- To name a file in a directory, use `dirName/fileName`
- Can concatenate directory names to form a path
  - foo/bar/baz/blip

### Special directory names
- . names the current directory
- .. names the enclosing directory
- / names the root directory (sort of)
- ~ names your home directory (sort of)

- What is `/foo/bar/../`?
- What is `/../`?

### Some basic commands
- `ls`
  - list current directory contents
  - `ls -l` for detailed listing
- `mkdir dirName..`
  - create one or more nested directories of given names
- `cd dirName`
  - change current directory to named one
    - can be a full path name, as with most commands
- `pwd`
  - print name of current directory
Viewing files

- `cat fileName...`
  - print out contents of one or more files
- `more fileName...`
  - same as `cat`, but only a page at a time
- `lpr fileName...`
  - print out a file onto the "current" printer
  - `lpr -PprinterName fileName` for a specific printer

Copying and moving files

- `cp fromName toName`
  - copy a file from `fromName` to `toName`
- `mv fromName toName`
  - move a file from one name to another (which shouldn't exist yet)
  - `cp fromName... dirName`
  - move a file one or more files to an existing directory (keeping same names)

Removing files and directories

- `rm fileName...`
  - remove one or more files
- `rmdir dirName...`
  - remove one or more (empty) directories
- `rm -r fileName...`
  - remove one or more files, and their contents if directories
  - Know what you're doing!
    - (What does `rm -r` do?)

Creating and editing files

- `emacs fileName`
  - emacs is a very powerful & customizable editor
  - lots of control-key commands
  - X-windows versions support mouse clicking and menu bars
  - worth learning; we'll study more later
- Under cygwin, can do notepad `fileName`

Finding out more about commands

- `man commandName`
  - prints out manual on `commandName`
    - many cool options on earlier commands!
- `man -k keyword`
  - prints out all manual page titles that include keyword
  - (What does `man man` do?)

Permissions

- Every file has an owner and a group
  - Owner is usually the person (login id) who created the file (see chown)
  - Group is the group that can share access to the file (see chgrp, groups)
- Every file has permissions, which specify whether owner/group/everyone can read/write/"execute" the file
  - execute for a directory: can look inside
  - (see chmod, ls -l)
Filename patterns

- Can name a bunch of files using a filename
  pattern
- Embedded in regular filenames
- Wildcards
  - * matches a sequence of 0 or more chars
  - ? matches exactly one char
- Expanded into multiple arguments, based on
  matching file/pathnames

cp test?.htm* ~/www

More filename patterns

- Character ranges
  - [a-eu] matches a lower-case vowel
  - [0-9A-Fa-f] matches a hexadecimal digit
- Sets
  - {foo,bar,baz} matches foo, bar, or baz
  - {foo,} matches foo or empty
  - can have patterns embedded in the list
- If more than one pattern, all combinations

ls {./usr{(/local)}{/Bb}in

Customizing the shell

- Shell has several ways to customize its behavior
  - Details depend on your default shell
  - I'll assume csh/tcsh; bash is a popular alternative

Shell variables

- set var, var=value, var=(value...)...
  - adds / changes one or more settings
  - set alone prints out all settings
  - unset var... removes one or more settings

set nonmatch history=100 autolist filec
set prompt = "%m: %~ %#h>"

- man csh or man tcsh to see all possibilities

Environment variables

- printenv
  - prints out all settings
- setenv varName value
  - adds / changes a setting (no "=")
- unsetenv varName
  - removes a setting

- Subtle distinction between vars and env vars

Printing out var values

- $varName as command argument is replaced with value of varName

- echo arg...
  - just prints out its arguments (silly, right?)

- echo $varName
  - prints out the value of varName
The path variable

- Unix finds commands using the *path* variable
  - set path = (... a list of directories ...)
  - (How to print out your current path?)
- When the shell sees
  command arg...
  it looks for an executable file named `command` in a directory on the path, searching in order, and then runs it.

Adding to the path

- Can add your own directories of commands by changing the path variable
  - Keep all the old directories!
- To add `~/bin` to the path [why the front?] :
  - set path = (~/bin $path)
- Now can put my programs in `~/bin`
  - mv myProg ~/bin
  - hash
    - tell the shell to recompile what programs are available
  - myProg myArgs... now works, from anywhere!

Saving customizations

- When you log in, shell automatically runs commands in the file `~/ .cshrc` (or `~/ .bashrc` or `~/ .profile` or ...)
- For any settings you want all the time (e.g. the expanded path setting), add them to your `.cshrc` file, and you'll get them automatically when you log in next time.

Quoting

- Sometimes want to stop shell from doing
  filename pattern expansions, or `$var` expansions, or argument splitting on spaces
- Can do this in several ways:
  - Surround with single quotes
    - turn off all expansions
  - Surround with double quotes
    - still allows `$var` expansions
  - Use \ on selected characters
    - disable any special meaning

Examples of quoting

- `cp t.txt "a file name with spaces.txt"`
  - 2 arguments
- `echo "\$path is $path"
  - 1 argument
- `echo \$path is $path
  - 2 arguments
- `echo $\$path is $path
  - 3 arguments

An advanced command: grep

- `grep regularExpression fileName`
  - search the named file(s) for all lines that match
    - anywhere the given *regular expression*, and
    - print them out
  - egrep, fgrep are variations that have slightly different regex languages
  - grep -- regEx fileName...
    - prints lines that dont match
  - Regular expressions are like filename patterns, but more powerful
  - Several Unix commands have similar *regular expression* sublanguages, so good to know
Regular expressions

- Like filename patterns, except different special characters
- . matches any character (like *)
- re* matches zero or more occurrences of the previous regular expression re
  - can use (...) to bracket a regex to repeat
  - or on some greps, {...}
- * regex is same as * filename pattern
- (What does a(b.c)*d match?)

Matching start or end of line

- ^ at the front of a regex means that the regex must start matching at the start of a line
- $ at the end ... at the end of a line
- grep '^[^]*$' *java
  - matches lines that are just

grep quiz

- Print out all the abstract class and interface declarations in some .java files
- Find all lines in the .java files that reference System.out.print or System.out.println
- Print all non-blank lines in a file

Another adv. command: sed

- sed -e command fileName...
  - sed can be used to perform edits to the input file(s), printing out the result
  - command is a special sed command
  - can have as many -e command/arguments as desired
  - can omit -e if only one command
- lots of possible script commands
  - [how to find out?]
  - we'll look at one: the s command

String replace using sed

- sed 's/regex/replacement/g' fileName
  - finds all occurrences of phrases matching regex in input file
  - replaces each with replacement
  - if leave g off, then only replace first match
  - / can be any character
Bound substrings
- Can remember parts of phrase matching regex, reuse them in replacement
  - \& refers to whole matched phrase
  - \1 ... \9 refer to corresponding matching subphrases inside parens
- sed 's/abstract class \(.*\) extends interface \1 implements/g' file.java

sed quiz
- Replace all occurrences of toString with ToString in the input file
- Extract and print all //style comments (just the comments!)

Another adv. command: find
- find dirName... options...
  - do recursive searching or processing of given directories and all the files & subdirectories they contain, based on options
  - options can be tests that decide whether to consider the file, or commands to perform on that file

Some find tests
- -name filenamePattern
  - only match files whose names match filenamePattern
- -type t \(\text{tis for f or d or ...}\)
  - only match files that are plain files \(f\) or directories \(d\) or ... \(\text{not, or, ...}\)
  - allow boolean combinations to be specified
  - (and is implicit connector)

Some find actions
- -print
  - print out the path name of the current file
- -exec command arg... \;
  - run the command
  - {} in args replaced with matching name
- -prune
  - don't recursively search this directory

find quiz
- Print out the path names of all files in current directory whose name is README
- Remove every file and directory whose name is tmp or temp or ends with ~
Redirecting output

- So far, commands have appeared to always print their results out to the screen
- Really, output goes to standard output (stdout), which defaults to the screen
  - There's also standard error (stderr), for any error messages, which also defaults to the screen
- It's easy to redirect stdout, e.g. to a file
  - Good if need to to save output for later
  - Good if want to use output as input file for another command (but more on this later)

Redirecting output to a file

- `command arg... > fileName`
  - Redirects command's stdout to `fileName`
- Overwrites `fileName` if it exists
  - Use `>>` instead to append to file
- Leaves stderr alone
  - Use `&` or `>&&` instead to redirect both stdout & stderr to the same file

Programs as stream processors

- Since output redirection is easy, many Unix programs defined to produce their output on stdout, and then let users decide what to do with it
- Likewise, many programs defined to take their input from `standard input (stdin)`, if no explicit file arguments are given
  - stdin defaults to the keyboard
  - can be redirected to a file using `<`
- Model: `stdin → program → stdout`

Pipelines

- To exploit this uniform input/output processing, can arrange sequences of programs in pipelines
- `stdin → cmd1 | cmd2 | ... | cmdN → stdout`
  - `grep regex * .java | more`
  - `ls -l | grep Jan | more`

Pipeline utilities

- Pipelining leads to lots of simple utilities that do one thing well that can be combined to create interesting effects
- Some sources:
  - `cat, echo, ls, find, diff, input file redirection`
- Some filters & processors:
  - `grep, sed, sort, uniq, tee, wc, head, tail`
- Some sinks:
  - `more, output file redirection`