

---

CSE 374

# Programming Concepts & Tools

Hal Perkins

Spring 2009

Lecture 3 – I/O Redirection, Shell Scripts

---

# Program input/output

---

- Old news: Programs take an array of strings as arguments
- Also: Programs return an integer (convention: 0 for “success”)
  - Previous command’s exit status is in shell variable \$?
- The shell also sets up 3 “streams” of data for the program to access:
  - stdin a.k.a. 0: an input stream
  - stdout a.k.a. 1: an output stream
  - stderr a.k.a. 2: another output stream
- The default shell behavior uses the keyboard for stdin and the shell window for stdout and stderr.
- Examples:
  - ls prints regular output to stdout and “No match” to stderr.
  - mail takes message body from stdin (waiting for C-d (“end of file”) to stop taking input).

# File redirection in detail

---

- Using arcane characters, we can tell the shell to use files instead of the keyboard/screen:
  - redirect input: `cmd < file`
  - redirect output, overwriting file: `cmd > file`
  - redirect output, appending to file: `cmd >> file`
  - redirect error output: `cmd 2> file`
  - redirect output and error output to file: `cmd &> file`
  - ...

See bash manual sec. 3.6 for other variations
- Useful special file: `/dev/null`
  - Immediate eof if read; data discarded if written

# Pipes

---

cmd1 | cmd2

- Change the stdout of cmd1 and the stdin of cmd2 to be the same, new stream!
- Very powerful idea:
  - In the shell, larger command out of smaller commands
  - To the user, combine small programs to get more usefulness
    - Each program can do one thing and do it well!
- Examples:
  - foo --help | less
  - djpeg me.jpg | pnmscale -xysize 100 150 | cjpeg > thumb.jpg

# Combining commands

---

- Combining simpler commands to form more complicated ones is very programming-like. In addition to pipes, we have:

`cmd1 ; cmd2` (sequence)

`cmd1 || cmd2` (or, using int result – the “exit status”)

`cmd1 && cmd2` (and, like or)

`cmd1 `cmd2`` (use output of `cmd2` as input to `cmd1`).

(Very useful for your homework. Note `cmd2` surrounded by backquotes, not regular quotes)

- Useless example: `cd `pwd``.
- Non-useless example: `mkdir `whoami`A`whoami``.

# (Non)-alphabet soup

---

- List of characters with special (before program/built-in runs) meaning is growing: ' ! % & \* ~ ? [ ] " ' \ > < | \$ (and we're not done).
- If you ever want these characters or (space) in something like an argument, you need some form of escaping; each of " ' \ have slightly different meaning.
- First approximation:
  - "stuff" treats stuff as a single argument but allows some substitutions for \$variables.  
example: `cat "to-do list" # filename with spaces(!)`
  - 'stuff' suppresses basically all substitutions and treats stuff literally.

# Shell as a programming language

---

- 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.
  - The shell interprets lots of funny characters differently, rather than pass them as options to programs.
- It’s actually even more complicated:
  - (two kinds of) variables.
  - some programming constructs (conditionals, loops, etc.)

# Toward Scripts...

---

- A running shell has a state, i.e., a current
  - working directory
  - user
  - collection of aliases
  - history
  - ...
- In fact, next time we will learn how to extend this state with new shell variables.
- We learned that source can execute a file's contents, which can affect the shell's state.



# Running a script

---

- What if we want to run a bunch of commands without changing our shell's state?
- Answer: start a new shell (sharing our stdin, stdout, stderr), run the commands in it, and exit.
- Better answer: Automate this process.
  - A shell script as a program (user doesn't even know it's a script).
  - Now we'll want the shell to end up being a programming language
  - But it will be a bad one except for simple things

# Writing a script

---

- Make the first line exactly: `#!/bin/bash`
- Give yourself “execute” permission on the file
- Run it
  - Probably need to precede filename with `./` if current directory isn’t normally searched for commands
- Note: The shell consults the first line:
  - If a shell-program is there, launch it and run the script (similar trick works for perl, python, etc.)
  - Else if it’s a “real executable” run it (more later).
- Example: `listhome`

# More expressions

---

- bash expressions can be:
  - math or string tests (e.g., `-lt`)
  - logic (`&&`, `||`, `!`) (if you use double-brackets)
  - file tests (very common; see Pocket Guide)
  - math (if you use double-parens)
- Gotcha: parens and brackets must have spaces before and after them!
- Example: `dcldls` (double `cd` and `ls`) can check that arguments are directories.
- Exercise: script that replaces older file with newer one
- Exercise: make up your own

# Accessing arguments

---

- The script accesses the arguments with  $\$i$  to get the  $i$ th one (name of program is  $\$0$ ).
  - Example: make thumbnail1
- Also very useful for homework: shift (manual Section 4.1)
  - Example: countdown
- We would like optional arguments and/or usage messages. Need:
  - way to find out the number of arguments
  - a conditional
  - some stuff we already have
  - Example: make thumbnail2

# Review

---

- The shell runs programs and builtins, interpreting special characters for filenames, history, I/O redirection.
- Some builtins like `if` support rudimentary programming.
- A script is a program to its user, but is written using shell commands.
- So the shell language is okay for interaction and “quick-and-dirty” programs, making it a strange beast.
- For both, shell *variables* are extremely useful.

# Preview: Variables

---

```
i=17 # no spaces
set
echo $i
set | grep i
echo $i
unset i
echo $i
f1-$1
```

- (The last is very useful in scripts before shifting, e.g., see homework.)
- Enough for your homework (arithmetic, conditionals, shift, variables, redirection, ...)
- Gotcha: using undefined variables (e.g., because of typo) doesn't fail (just the empty string).