CSE 303:
Concepts and Tools for Software Development

Hal Perkins

Autumn 2008

Lecture 3— I/O Redirection, Shell Scripts

Where are We

- A simple view of the system: files, users, processes, shell
- Lots of small useful programs; more to come
- An ever-more-complicated shell definition:
  - Filename expansion
  - Command-line editing
  - History expansion
  - I/O redirection
  - Programming constructs
  - Variables

Simple view of input/output

- Old news: Programs take an array of strings as *arguments*
- Also: Programs return an integer (convention: 0 for "success")

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 files `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

Using arcane characters, we can tell the shell to use files instead of the keyboard/screen (Bash Manual, Section 3.6):

- 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*
- ...

Examples:

- How I get the histories for the web page.
- `ls` uses `stdout` and `stderr`.
- Using `cat` to copy information to/from files.

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 > me_thumb.jpg`

cat and redirection

Just to show there is some math underlying all this nonsense, here are some fun and useless equivalences (like $\mathbf{1} \cdot \boldsymbol{y} = \boldsymbol{y}$):

- `cat` $\boldsymbol{y}$ = `cat <` $\boldsymbol{y}$
- $\boldsymbol{x}$ `<` $\boldsymbol{y}$ = `cat` $\boldsymbol{y}$ `|` $\boldsymbol{x}$
- $\boldsymbol{x}$ `| cat` = $\boldsymbol{x}$

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‘`.

Note: Previous line's exit status is in `$?`.

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.

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

Note: The shell consults the first line:

- If a shell-program is there, launch it and run the script
- Else if it's a "real executable" run it (more later).

Example: `listhome`

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`

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: `dcdls` (double `cd` and `ls`) can check that arguments are directories.

Exercise: script that replaces older file with newer one

Exercise: make up your own

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.

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).