CSCI 330
Shell Basics

Jon Lehuta

August 17, 2020

Shell Basics - Outline

UNIX Command Interpreters

The program that interprets our commands and allows us to interact with the operating system is called the *shell*.

- Families:
  - Bourne shell (`sh`)
    - Developed as part of original, commercial UNIX
  - C shell (`csh`)
    - Developed as part of free, academic UNIX
- Standard:
  - Every UNIX system has a “Bourne shell compatible” shell

Bourne shell family

- `sh`: Original Bourne shell
  - Written 1978 by Steve Bourne
  - Part of commercial UNIX
- `ash`: Almquist shell
  - BSD-licensed replacement of `sh`
- `bash`: Bourne-again shell
  - GNU replacement of `sh`
- `dash`: Debian Almquist shell
  - Scripting shell in Ubuntu (small, fast)
- Others (`ksh`, `zsh`, `busybox`)

For this class, we will be using `bash`. Most of these features will also work in the other shells.

`bash` Shell Basics

- Customization
  - Variables
  - Prompt and aliases
- Startup and initialization
- Command line behavior
  - History
  - Quoting
  - I/O Redirection and pipe
  - Background
  - Directory stacks
  - `eval` command

Customizations

- via command line options
  - rarely done (shell usually run automatically)
  - instead: initialization script autoruns
  - `~/.bash_profile` if login session shell
  - `~/.bashrc` if invoked from command line
- via variables
- path to find executables
- terminal settings: speed, size
- custom prompt
- command aliases

Variables

The shell can store values in named variables.

Variables are always stored as strings (or arrays of strings.), but you can type numbers as strings if needed.

To set variable

```
% varname=value
```

The shell will substitute in the value of a variable when it is found in an expression after the `$`.

For example, to display `varname`'s value from above,

```
% echo $varname
```

Setting variables in the shell

Syntax: `varname=value`

Example:

```
% speed=fast
% echo "Today we go: $speed"
Today we go: fast
% speed="very fast"
% echo "Today we go: $speed"
Today we go: very fast
```

Notice that there are no spaces around the `=`. If the value has spaces, we'll need quotes or backslashes to escape them.

Northern Illinois University

Variable Scope

Valid for duration of current shell invocation

Variable can be exported into environment:

```
  export fast
```

You can set variables temporarily for individual commands. For example,

```
% VARIABLE=value command
```

will run `command`, and if `command` tries to look for the value of `VARIABLE`, it will get the `"value"` value set on this line.

Once command stops running, `VARIABLE` will have whatever value it had before we ran that line.

Predefined Shell Variables

Name	Value
`HOME`	absolute path to your home directory
`PATH`	list of directories to search for commands
`MAIL`	absolute path to your system mailbox
`USER`	your username
`SHELL`	absolute path to your *login shell*
`TERM`	format of terminal being used
`PWD`	current working directory
`EDITOR`	default editor to use when needed
`DISPLAY`	where GUI apps should show (X server)

Predefined Environment Variables

Name	Value
`BASH`	full path of `bash` instance in use
`RANDOM`	a new, random integer from 0 to 32,767
`HOSTNAME`	hostname of current system eg. `turing`
`HOSTTYPE`	platform type, `i386`, `x86_64`, `ppc`
`PS1`	text to be used as primary prompt
`HISTSIZE`	number of command lines to remember
`?`	return code (status) of last command
`$`	process id (pid) of current shell

`printenv` displays all environment variables

Example: PATH variable

PATH should contain a colon-separated list of directories

When you type a command without specifying the exact path the executable that implements that command may be found, the shell will check the paths in PATH (in order) to try to locate it, so it can run the command.

If the command is not found in any of these directories, it will tell you "`Command not found.`"

If a command runs when you type its name, it must be in the PATH. To find out where it was found, you can use the `which` command.

```
% which ls
/bin/ls
```

Working with PATH

```
### To show the search path, use echo with its value ($PATH)
% echo $PATH
/usr/sbin:/usr/bin:/sbin:/bin:/usr/games

### To add to the beginning of the system path
% PATH=/usr/local/bin:$PATH
% echo $PATH
/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games

### To add to the end of the system path
% PATH=$PATH:~/bin
% echo $PATH

usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/home/student/bin
```

bash Shell Prompt

There are several prompts that the shell will show you. Their appearance can be changed by setting environment variables.

The main shell's appearance can be set via the `PS1` variable.

Example:

```
% PS1="type something already! > "
type something already! >
```

There are four types of prompt in bash, and each has its own variable

`PS1`	main prompt, whenever the shell is ready for a command
`PS2`	secondary prompt, you will see when entering multiple lines
`PS3`	used for `select` command's prompt
`PS4`	used when tracing (debugging) at shell

`bash` Shell Prompt

`PS1` is evaluated each time the shell prints the prompt, so you can have it show environment variables, and/or use the following

`\w`	current work directory
`\h`	hostname
`\u`	username
`\!`	history event number
`\d`	date
`\t`	time
`\a`	ring the “system bell”

Example:

```
% PS1="\u@\h-\!: "
z123456@turing-22:
```

Shell Aliases

The `alias` command aAllows you to assign an *alias*, to command(s). An *alias* is another name for something, and when you type it as a command, it will run the command it is an alias for.

To check current aliases:

```
% alias
```

To set alias:

```
% alias ll="ls -al"
```

To remove alias:

```
% unalias ll
```

To Create an Alias

On the command line

- via a text file
  - enter alias commands into text file
  - execute that text file with `source` or `.` command, which reads and executes the text into the context of the current shell

Aliases have the same scope as shell variables, so they go away when you log out.

You need to add them again when you log back in if you'd like them to stick around.

You can do this by adding them in the file that runs when the shell starts (`~/.bashrc` or `~/.bash_profile`).

Command line behavior

- History (`!`)
- Sequences (`;`)
- Command Substitution (`` ` ``, `$()`)
- I/O redirection and pipe (`>`, `<`, `>>`, `<<`, `|`)

Shell History

The shell keeps a record of previously entered commands so that they can later be:

- re-executed
- edited

Commands are saved

- per session
- per user

Size of history (in lines) can be set via shell variables:

```
HISTSIZE=500
HISTFILESIZE=100
```

Northern Illinois University

Shell History

Each command in history has a sequential event number

to view the history buffer:

```
history [-c] [count]
```

- If no options are supplied, list all history
- `-c` clears history

Shell History

You can re-execute history events:

- By last command
  - `% !!`
- By the event number
  - `% !5`
- By the number relative to current event (ago)
  - `% !-3`
- By the text of the command used
  - `% !ls`

The shell replaces these with the full text of the history event command line. Anything you type after them will be appended to what was typed on the repeated event.

Command line editing

As you are typing a command at the shell, you can work with the history with these keys:

- Up arrow:
  - Replace current command line with previous command in history
- Down arrow:
  - Replace current command line with next command in history (when there is one)
- Left/right arrows:
  - Move cursor horizontally
- Backspace:
  - Remove character after cursor.
- Delete:
  - Remove character before cursor.
- Tab:
  - Try to autocomplete current command or file name

Command Sequence

Allows series of commands to be specified on a single line.

Commands are separated by a semicolon. `;`

Example:

```
% date; pwd; ls
```

Command Substitution

A command surrounded by backticks (`` `command` ``) is run before the rest of the command line.

Whatever the inner command outputs to standard output is captured, and the shell replaces the backticks and everything inside of them with that captured output. Newlines in the output are replaced by spaces.

Examples:

```
% echo Bob
Bob
% echo `echo Bob`
Bob
% echo Today is `date` and it is the greatest
% ls -l `which passwd`
-rwxr-xr-x 1 root wheel 68656 May  4 02:04 /usr/bin/passwd
% var=`whoami`; echo $var
```

Command Substitution

Another form of command substitution: `$(command)`

It works the same as the backticks, but allows for nesting of commands.

Examples:

```
% echo "User $(whoami) is on $(hostname)"
User z123456 is on turing

% echo "Today is" $(date)
Today is Fri Jul 17 08:06:28 CDT 2018
```

Uses for command substitution

Command substitution is useful if you need to run a command using data that another command prints out, or for putting text around text that another command prints out on a line alone.

- Automatically appending the date to filenames:

```
% cp "$filename" "$filename$(date +%F)"
```

- As a more advanced example, lots of libraries for UNIX systems are tracked by a program called `pkg-config`, and that program can be used to print out the compiler flags needed to compile with a given library. Command substitution is frequently used on `pkg-config` to automate compilation.

Northern Illinois University

Output Redirection (>)

Syntax: `command > file`

Standard output stream from `command` written to `file`, instead of to the terminal.

If output file already exists, it is overwritten by the new data. If not, the file is created.

Examples:

```
% ls > listing
% cat file > filecopy
% echo "This goes in the file." > file
```

Input Redirection (<)

Syntax: `command < file`

Command will take its standard input (`cin`) from `file`, instead of from terminal.

Example:

```
% tr '[a-z]' '[A-Z]' < listing
```

This can be useful if you're testing a program that asks a lot of questions on `cin`. You can type the answers into a file and have the shell answer for you.

Redirecting Input *and* Output

It is acceptable to redirect input and output simultaneously.

```
% tr [A-Z] [a-z] < r.in > r.out
```

Add sequence operator, and we can have the output of the first command go into a file, which becomes the input to the next command:

```
% ls > temp.txt; wc < temp.txt
```

Our next shell feature allows us to do the above, without the intermediate file:

```
% ls | wc
```

The pipe (|) operator means that the output from the program on the left goes directly into the input of the program on the right.

Output Redirection (append mode)

Syntax: `command >> file`

Works like `>` did, but adds output of command at the end of an existing file, instead of overwriting what was there. If file does not exist, shell still creates it.

Examples:

```
% date > usage-status
% ls -l >> usage-status
% du -s >> usage-status
```

Here Document

The "Here Document" operator for the shell is indicated with `<<`.

Like input redirection with the single `<`, it is affecting the standard input of the command run

Unlike simple input redirection, the data does not come from a file, but is instead supplied at the command line.

Syntax: `command << LABEL`

- `LABEL` can be whatever you'd like, but should not appear in input data
- when you hit enter, the shell will continue to prompt you for more lines
- whatever you type on these new prompts becomes the data sent to the program
- input stops when your `LABEL` appears on a new line by itself

Here Document example

Example:

```
# wc counts lines, words, characters of standard input
% wc << DONE
> line one
> line two
> DONE
    2   4   18
```

File Descriptors

On a UNIX system, every program runs in a process, and each process has its own set of file descriptors – positive integers that indicate open files.

Even before you explicitly open a file, a process starts with three file descriptors already open.

0	standard input (`cin` in C++)
1	standard output (`cout` in C++)
2	standard error (`cerr` in C++)

The shell can use these numbers to control input/output redirection:

```
% runprogram 1> stdout.log 2> error.log 0< input
```

Redirection syntax

- Output:
  - standard output redirection:
    - `> filename`
    - `1> filename`
  - standard error redirection:
    - `2> filename`
  - redirect both
    - `2>&1 >filename`
    - `&> filename`
    - `>& filename`
- Input:
  - standard input is the only option
    - `<`
    - `0< file`

Example:

```
% cat hugo > /tmp/one 2>&1
% cat hugo &> /tmp/one
```

C++ Program for Redirection

```
/**| noisy.cc |****************************************
 * This example program does something with each of the
 * three available streams.
 **| compile with: g++ noisy.cc -o noisy |***************/
#include <iostream>
#include <string>
using std::cin;  using std::endl;
using std::cout; using std::cerr;

int main() {
  std::string s;
  cin >> s; // read from stdin
  cout << "This goes to stdout: got " << s << endl;
  cerr << "This goes to stderr: got " << s << endl;
  return 0;
}
```

I/O Redirection Examples

```
% ./noisy
bleep
This goes to stdout: got bleep
This goes to stderr: got bleep
% ./noisy > output.txt
bloop
This goes to stderr: got bloop
% cat output.txt
This goes to stdout: got bloop
./noisy < output.txt
This goes to stdout: got This
This goes to stderr: got This
% ./noisy 2> error.txt < output.txt
This goes to stdout: got This
```

Background/Foreground

Normally, when we run a program, it runs in the *foreground*. This means that the shell won't prompt for another command until the program finishes, and things we type go to the program on standard input.

It is possible to run a program in the *background*. When you do this, the program does run, but the shell is immediately ready for more commands. When you type, the shell will get the keystrokes.

To run a program in the background, put an ampersand (&) at the end of the command line.

As an example, let's say we want to move a lot of of big files to another disk (slow), but want to be able to do other things while we wait:

```
% mv bigfiles/* /mnt/usbdrive/ &
```

Directory Stacks

We have already learned how to change our current working directory with `cd`.

Sometimes we may want to go take care of something quickly in another directory, and then return to the place we were.

This can be a pain with just `cd`, but with directory stacks, it is easier.

Instead of `cd directory`, we can use `pushd directory`. This puts the current directory on top of the directory stack and changes to the new one.

When we'd like to return, `popd` changes to the directory that's on top of the directory stack and then pops it off the stack.

`eval` Command

The `eval` command runs its command line arguments as if they had been typed at the shell on their own

```
% echo Hello
Hello
% eval echo Hello
Hello
```

At first glance, it doesn't seem very useful, but it can be useful sometimes as a "glue" to make things work in scripts.