The Bash Shell and You

Nathan Jarus

- Slides: `http://web.mst.edu/~nmjxv3/articles/shell.pdf`
- Files: `http://web.mst.edu/~nmjxv3/articles/shell/`

October 3, 2017

What's so cool about shells?

What's so cool about shells?

- You can find 'em on the beach
- They look nice
- Cool ocean sounds
- Sometimes have crabs inside!

What's so cool about shells?

- Pretty fast for some tasks
- Works well over a slow internet connection
- Can construct complex tools out of simple ones
- Easy to automate tasks!

What *is* a shell, anyway?

- A shell takes commands, runs them, and shows you the output
- Linux has a bunch of shells: `bash`, `zsh`, `dash`, `csh`, `tcsh`, ...
- Shells usually provide some tools for connecting programs together, running multiple programs, &c.

What *is* a shell, anyway?

- A shell takes commands, runs them, and shows you the output
- Linux has a bunch of shells: `bash`, `zsh`, `dash`, `csh`, `tcsh`, ...
- Shells usually provide some tools for connecting programs together, running multiple programs, &c.

Is PuTTY a shell?

What *is* a shell, anyway?

- A shell takes commands, runs them, and shows you the output
- Linux has a bunch of shells: `bash`, `zsh`, `dash`, `csh`, `tcsh`, ...
- Shells usually provide some tools for connecting programs together, running multiple programs, &c.

Is PuTTY a shell? Nope, it's a terminal!

- Terminals connect your keyboard and screen to your shell
- They run on your machine and connect to a shell (either on your machine or somewhere else)
  - Linux: `xterm`, `gnome-terminal`, &c.
  - Mac: Terminal.app
  - Windows: PuTTY
- Can connect to remote shells via `ssh` (secure shell)

Navigating your filesystem

- `pwd`: Print Working Directory

Navigating your filesystem

- `pwd`: Print Working Directory
- `cd`: Change Directories
  - `cd` with no arguments takes you to your home directory
  - `cd -` takes you to the last directory you were in

Navigating your filesystem

- `pwd`: Print Working Directory
- `cd`: Change Directories
  - `cd` with no arguments takes you to your home directory
  - `cd -` takes you to the last directory you were in
- `ls`: List (files and directories)
  - `-l` Display a detailed list of information about each file
  - `-h` Display human-readable file sizes
  - `-a` Display all files, even hidden ones (files that start with a `.`)

Navigating your filesystem

- `pwd`: Print Working Directory
- `cd`: Change Directories
  - `cd` with no arguments takes you to your home directory
  - `cd -` takes you to the last directory you were in
- `ls`: List (files and directories)
  - `-l` Display a detailed list of information about each file
  - `-h` Display human-readable file sizes
  - `-a` Display all files, even hidden ones (files that start with a `.`)

Some neat tricks:

- `ls *.txt`
- `ls **/*.cpp`

Scooting files around

WARNING: These programs will happily destroy all your files if you ask them to

Scooting files around

WARNING: These programs will happily destroy all your files if you ask them to

- `mv`: Move (or rename) files
  - `-i`: Interactively ask before overwriting files

Scooting files around

WARNING: These programs will happily destroy all your files if you ask them to

- `mv`: Move (or rename) files
  - `-i`: Interactively ask before overwriting files
- `cp`
  - `-r`: Recursively copy directories
  - `-i`: Interactively ask before overwriting files

Scooting files around

WARNING: These programs will happily destroy all your files if you ask them to

- `mv`: Move (or rename) files
  - `-i`: Interactively ask before overwriting files
- `cp`
  - `-r`: Recursively copy directories
  - `-i`: Interactively ask before overwriting files
- `rm`
  - `-f`: Forcibly remove files (even if write-protected)
  - `-r`: Recursively remove directories
  - `-i`: Interactively ask before overwriting files

Scooting files around

WARNING: These programs will happily destroy all your files if you ask them to

- `mv`: Move (or rename) files
  - `-i`: Interactively ask before overwriting files
- `cp`
  - `-r`: Recursively copy directories
  - `-i`: Interactively ask before overwriting files
- `rm`
  - `-f`: Forcibly remove files (even if write-protected)
  - `-r`: Recursively remove directories
  - `-i`: Interactively ask before overwriting files

Another neat trick: `mv bob.{coo,cpp}`

Help!

- `help`: Help with built-in Bash commands and features
- `man`: Help with **E V E R Y T H I N G**
  - Scroll with arrow keys, j/k, or PgUp and PgDn
  - q quits
  - To search for something: `/search-term` Enter
  - n goes to next match; N goes to previous match
  - h shows more navigation hints

Looking at stuff

- `cat`: Put the contents of one or more files on the screen
- `less`: Display the contents of a file one page at a time

Looking at stuff

- `cat`: Put the contents of one or more files on the screen
- `less`: Display the contents of a file one page at a time
- `head` and `tail`: Display the first or last ten lines of a file

Looking for stuff

- `find`: Find files in a directory (and do stuff to them)
  - The first argument is a directory to search in
  - After that you can specify things to search for:
    - `-name`: Search by name or glob
    - `-type`: Search for `f`iles or `d`irectories
  - After that you can specify an action:
    - `-ls`: Show file output like `ls -l` does
    - `-delete`: Delete files!
    - `-exec`: Execute a command

Looking for stuff

- `find`: Find files in a directory (and do stuff to them)
  - The first argument is a directory to search in
  - After that you can specify things to search for:
    - `-name`: Search by name or glob
    - `-type`: Search for `f`iles or `d`irectories
  - After that you can specify an action:
    - `-ls`: Show file output like `ls -l` does
    - `-delete`: Delete files!
    - `-exec`: Execute a command
- `grep`: Search for stuff inside files
  - `-i`: Perform case-insensitive match
  - `-v`: Invert the match (print lines that don't match)
  - `-C 5`: Show 5 lines of context around matches

Standard input and output

Every program has one "input stream" (called `STDIN`) and two "output streams" (called `STDOUT` and `STDERR`). In C++, `STDIN` is connected to `cin`, `STDOUT` to `cout`, and `STDERR` to `cerr`.

Typically `STDIN` reads from your keyboard and `STDOUT` (and `STDERR`) write to the screen.

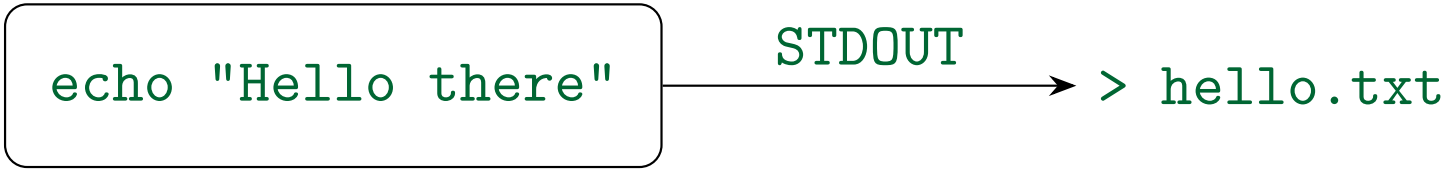
Writing output to files

You can use > and >> to redirect a command's output to a file instead!

Writing output to files

You can use `>` and `>>` to redirect a command's output to a file instead!

```
echo "Hello there" > hello.txt
```



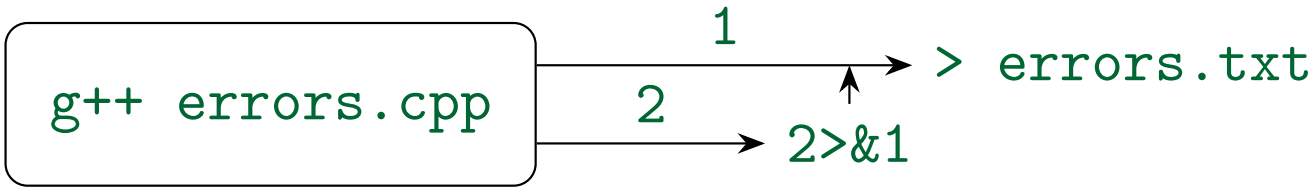
Capturing STDERR

Typically, you want to capture BOTH standard output and standard error. To do this, redirect standard error into standard output with the incantation `2>&1`.

Capturing STDERR

Typically, you want to capture BOTH standard output and standard error. To do this, redirect standard error into standard output with the incantation `2>&1`.

```
g++ errors.cpp > errors.txt 2>&1
```



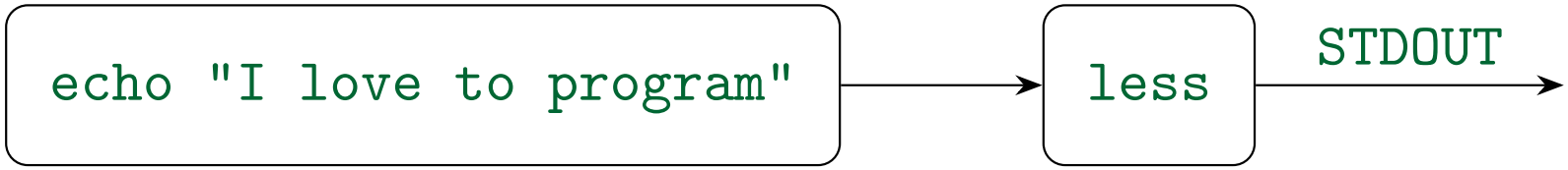
Writing output to other programs

You can also redirect one program's `STDOUT` to another program's `STDIN`! This is done with the pipe (`|`) character.

Writing output to other programs

You can also redirect one program's `STDOUT` to another program's `STDIN`! This is done with the pipe (`|`) character.

```
echo "I love to program" | less
```



Shell script anatomy

```
#!/bin/bash

g++ *.cpp -o program
./program
```

A shell script is comprised of two things:

1. A "shebang" line: starts with `#!`; contains the command that executes the script
2. A bunch of bash commands

Variables and Arguments

- To assign a value to a variable: `num_cows=5` (note: NO SPACES)
- To use a variable: `$num_cows`

Variables and Arguments

- To assign a value to a variable: `num_cows=5` (note: NO SPACES)
- To use a variable: `$num_cows`

Command-line arguments live in numbered variables:

- `$0`: The name of the program
- `$1`: The first argument
- `$2`: The second argument (and so on and so forth)

Variables and Arguments

- To assign a value to a variable: `num_cows=5` (note: NO SPACES)
- To use a variable: `$num_cows`

Command-line arguments live in numbered variables:

- `$0`: The name of the program
- `$1`: The first argument
- `$2`: The second argument (and so on and so forth)

```
#! /bin/bash

g++ *.cpp -o $1
./$1
```

Checking for failures

Our script has two problems right now:

1. If our code doesn't compile, we probably don't want it to run!
2. If the user doesn't offer a program name, the error message is not very nice

Checking for failures

Our script has two problems right now:

1. If our code doesn't compile, we probably don't want it to run!
2. If the user doesn't offer a program name, the error message is not very nice

More special variables:

- `$?`: The return value of the last program run
- `$#`: The number of command-line arguments passed

Checking for failures

```
#! /bin/bash
progname="program"

if [[ $# -ge 1 ]]; then
        progname=$1
fi

g++ *.cpp -o $progname

if [[ $? -eq 0 ]]; then
        ./$progname
fi
```

Doing stuff to a bunch of files

Let's pretend we don't know about `find` for a second. How would we make a backup copy of every one of our shell scripts?

Doing stuff to a bunch of files

Let's pretend we don't know about `find` for a second. How would we make a backup copy of every one of our shell scripts?

With a for loop!

```
#! /bin/bash

for file in *.sh; do
        echo "Copying $file to $file.bak"
        cp $file $file.bak
done
```

Looping over command-line arguments

Let's print out each command line argument on its own line:

```
#! /bin/bash

for arg in $@; do
        echo $arg
done
```

Running commands from anywhere

These shell scripts are great, but we always have to give `bash` the exact path to the script so it knows what to run. How can we fix this?

Running commands from anywhere

These shell scripts are great, but we always have to give `bash` the exact path to the script so it knows what to run. How can we fix this?

When you ask it to run a command, `bash` looks through all the directories listed in a special variable named `PATH`. We can add our own directory to this!

```
PATH=~/bin:$PATH
```

Running commands from anywhere

These shell scripts are great, but we always have to give `bash` the exact path to the script so it knows what to run. How can we fix this?

When you ask it to run a command, `bash` looks through all the directories listed in a special variable named `PATH`. We can add our own directory to this!

```
PATH=~/bin:$PATH
```

To get this modification to stick, we need to run this command every time we start our shell. Fortunately, `bash` runs the `/.bashrc` script every time you start a new `bash` shell!

Running commands from anywhere

These shell scripts are great, but we always have to give `bash` the exact path to the script so it knows what to run. How can we fix this?

When you ask it to run a command, `bash` looks through all the directories listed in a special variable named `PATH`. We can add our own directory to this!

```
PATH=~/bin:$PATH
```

To get this modification to stick, we need to run this command every time we start our shell. Fortunately, `bash` runs the `/.bashrc` script every time you start a new `bash` shell!

For more variables that control how `bash` works, see `help variables`.

Making new commands the quick ‘n easy way

- `alias name=command`: Make a shorthand name for an existing command
- Bash functions: `function-name() { commands }`

Making new commands the quick ‘n easy way

- `alias name=command`: Make a shorthand name for an existing command
- Bash functions: `function-name() { commands }`

```
rungcc() {
        progname="program"

        if [[ $# -ge 1 ]]; then
                progname=$1
        fi

        g++ *.cpp -o $progname

        if [[ $? -eq 0 ]]; then
                ./$progname
        fi
}
```

Where to from here?

Take CS 1585! `http://web.mst.edu/~nmjxv3/cs1001/`

- List of Bash Commands:
  `https://ss64.com/bash/`
- Bash Reference Manual:
  `https://www.gnu.org/software/bash/manual/`
- All About Pipes:
  `http://www.linfo.org/pipe.html`
- Software Carpentry Shell Tutorial:
  `http://swcarpentry.github.io/shell-novice/`
- Bash Tutorial:
  `http://tldp.org/LDP/Bash-Beginners-Guide/html/`

Pictured: Annie (left) and Lion (right)