Programming in Bash for Fun and Profit

Christopher W. Pitts

July 14, 2017

Disclaimer

The views in this presentation are entirely my own, and *in no way* represent any sort of official or unofficial endorsement by Sandia National Laboratories or NTESS.

Follow Along?

https://github.com/cwpitts/bash-presentation

About Me

- From Albuquerque, New Mexico
- Raging *Star Wars* fan
    - Still bitter about Disney and the Expanded Universe
- Studied computer science at Brigham Young University
- Software Systems Engineer at Sandia National Laboratories
- Ik spreek Hollands (ook Vlaams)!
- Happily married

Overview

1. Intro
2. Common Pitfalls
3. Features
4. Useful Applications Of Bash
5. Conclusion

Why Are You Here?

Why Are You Here?

- Use it at work

Why Are You Here?

- Use it at work
- Have to maintain someone else's Bash code

Why Are You Here?

- Use it at work
- Have to maintain someone else's Bash code
- I read the Bash script that installs Salt, and it gave me nightmares

What Is Bash?

What Is Bash?

- A shell

What Is Bash?

What Is Bash?

- A shell
- A scripting language

What Is Bash?

What Is Bash?

- A shell
- A scripting language
- The verb that describes what your head does to the desk after the first ten minutes of trying to learn the scripting language

What Is Bash?

What Is Bash?

- Bourne Again SHell
  - sh-compatible shell
  - incorporates useful features from Korn and C shells
  - default shell in GNU/Linux

What Is Bash?

No, Really, What **is** Bash?

- Bash is a programming language
    - Data types?
        - Nope!
        - Bash has untyped variables
    - Containers?
        - Sequential arrays
        - Associative arrays (a.k.a. maps, dictionaries)
    - Flow control?
        - Conditional statements
        - Loops

Bash Is Dangerous!

Bash is dangerous, *if used unwisely*.

Be careful out there!

For example, don’t do this:

```
curl | sudo bash
```

Looping Over 'ls'

Don't do this!

```
#!/bin/bash

#
for file in $(ls)
do
    printf "%s\n" "${file}"
    cat "${file}"
    printf "\n"
done
```

Looping Over 'ls'

Looping over the output of **ls** is considered *fragile* and *dangerous*

- Fragile: Special characters (newlines, spaces, etc.) can break the loop vey easily
- Dangerous: Special characters can cause unintended consequences

Looping Over ’ls’

Try this instead:

```
#!/bin/bash

for file in *.txt
do
    printf "%s\n" "${file}"
    cat "${file}"
    printf "\n"
done
```

Looping Over 'ls'

Globbing is safer

- Regex matching allows for finer control
- Special characters won't break the loop

Word Splitting

What will this code do?

```
#!/bin/bash

var="This is a sentence."

printf $var
```

Word Splitting

Let's talk about **IFS**.

IFS is the "**I**nternal **F**ield **S**eparator", the delimiter that Bash uses to separate words, array entries, and arguments.

You'll notice that only the first word of the sentence was printed. This is because $var expands to the full sentence, so you end up with something equivalent to this:

```
$ printf This is a sentence.
```

Word Splitting

If we look at the manual page for **printf**:

PRINTF(1)   User Commands   PRINTF(1)

NAME
printf - format and print data

SYNOPSIS
printf FORMAT [ARGUMENT]...

Word Splitting

The moral of the story: *always* quote your variables (and use **printf** correctly):

```
#!/bin/bash

var="This is a sentence."

printf "%s" "${var}"
```

Double Or Single Quotes?

```
#!/bin/bash
var="foobar"

# This will not do what we want
printf '%s\n' '${var}'

# This will
printf "%s\n" "${var}"

# So will this, as it happens
printf '%s\n' "${var}"
```

Double Or Single Quotes?

Single quotes

- Literal strings
- No parameter expansion (the '$varname')

Double quotes

- Interpolated strings
- Parameter expansion
- Interpreted characters:
  - $ (parameter expansion and subshell)
  - ` (old subshell syntax)
  - \ (escape sequences)

Variables

```
#!/bin/bash

# Correct
var=1

# Incorrect (whitespace not allowed)
var = 1

# Access values with $
var2=${var1}

printf "%d\n" "${var2}"
```

Math

```bash
#!/bin/bash

x=10

# To do math, use ((<math>))
((x--))
((x=x-1))

# Or <var>=$((<math>))
x=$((x-1))

# Or let
let x-- # That's a double dash there
printf "%d\n" "${x}"
```

Dynamic Typing

```
#!/bin/bash

# x is a string
x="foobar"
printf "%s\n" "${x}"

# x is now a number!
x=7
printf "%d\n" "${x}"
# Or a string?
printf "%s\n" "${x}"
```

Undeclared Variables

```
#!/bin/bash

x="foobar"

# We already know what this does
printf "|%s|\n" "${x}"

# But what happens here?
printf "|%s|\n" "${y}"
```

Input/Output

Two main kinds of input in Bash

- Command-line arguments
- Command output

Two main types of output in Bash

- Return code
- stdout

Command-line arguments

```
#!/bin/bash

# Arguments are passed in in an array
# Arbitrary access with ${<index>}

printf "The first argument: %s\n" "${1}"
printf "The second argument: %s\n" "${2}"
printf "Did we get a third? %s\n" "${3}"

# Get all of the values in an array with ${@}
printf "%s\n" "${@}"
```

Command output

```
#!/bin/bash

# Use $(<command>) to capture command output
# Note the distinct lack of spaces
c=$(ls)
printf "%s\n" "${c}"
```

Return code

```bash
#!/bin/bash

declare -a arr=("manjaro" "arch" "ubuntu" "fedora")
filename="example.txt"

for i in ${arr[@]}
do
    printf "%s\n" "${i}" >> ${filename}
done

if grep -q "${1}" ${filename}
then
    printf "Found %s!\n" "${1}"
fi

rm ${filename}
```

stdout

```bash
#!/bin/bash

function even_or_odd()
{
    if (( $((${1} % 2)) == 0 ))
    then
        printf "Even!\n"
    else
        printf "Odd!\n"
    fi
}

res=$(even_or_odd ${1})
printf "%s\n" "${res}"
```

Flow control

- Conditionals
- Loops
  - For loops
  - While loops
  - Until loops

Conditionals

```
#!/bin/bash

target=5
# Getting input from command line arguments
num=${1}

# Comparisons are [ <statement> ]
if [ ${num} -gt ${target} ]
then
    printf "greater!\n"
elif [ ${num} -lt ${target} ]
then
    printf "less!\n"
else
    printf "equal!\n"
fi # Note we end with 'fi'
```

For loops

Pythonic **for** loops:

```
#!/bin/bash

# Python-style for x in y loop
# $(seq 10) expands to 1 2 3 4 5 6 7 8 9 10
for i in $(seq 10)
do
    printf "%d\n" "${i}"
done
```

For Loops

C/C++-style **for** loops:

```
#!/bin/bash

# C/C++-style for loop
for ((i=0; i < 10; i++))
do
    printf "%d\n" "${i}"
done
```

While loops

```
#!/bin/bash

k=1
while ((k < 10))
do
    printf "%d\n" "${k}"
    ((k=k+1)) # Note the ((math)) setup here
    # Could also do let k++
done
```

Until loops

```
#!/bin/bash

k=0
until ((k == 10))
do
    printf "%d\n" "${k}"
    ((k++))
done
```

Functions

Functions in Bash are pretty straightforward.

```
#!/bin/bash

# Could also do: function f
# Parens make it look nice
function f()
{
    printf "function f got: %s\n" "${1}"
}

# Call like it's on the command line
# <function> <arguments>
f "${1}"
```

Can You Do Anything Useful In Bash?

Yes!

Caveat: Bash is great for gluing things together, but you wouldn't want to write a webserver in it.

Mass Renaming

You could write a mass-rename utility

```bash
#!/bin/bash

oldExt="${1}"
newExt="${2}"

# Use this regex globbing instead of $(ls)
for file in *.${oldExt}
do
    # Get base filename (strip extension)
    f=$(printf "%s" "${file}" | cut -d '.' -f 1)

    # Move to new file and change extension
    mv "${file}" "${f}.${newExt}"
done
```

Slackbot

How about a Slackbot?

```
#!/bin/bash

msg="${1}"
slackteam="${2}"

curl -X POST --data-urlencode\
     "payload={\"text\":\"${msg}\"}"\
     "${slackteam}"
```

Customize Your .bashrc

Supercharge your command line by tweaking your .bashrc file

Customize Your .bashrc

Supercharge your command line by tweaking your .bashrc file

- Define custom functions to use on the command line
- Define often-used flags for a command as an *alias*
- Make your prompt way more interesting

Customize Your .bashrc

Custom functions

```
# Changing behavior of cd
function cd
{
    builtin cd "$@" && ls
}
```

Customize Your .bashrc

Aliases

```
#!/bin/bash

# Shorthand for commands
alias a='ls'

# Really good for shortening commands with flags
alias q='ls -slap'

# Or renaming them
alias bat='acpi'
```

Customize Your .bashrc

Aliases are also good for adding “default” flags

```
# Always use the -p for mkdir
alias mkdir='mkdir -p'

# Always use -l, -h, -a for ls
alias ls='ls -lah'
```

Customize Your .bashrc

Custom prompt

```
# PS1 is the "prompt variable"
export PS1="\W >>> "
```

Customize Your .bashrc

Some useful escape sequences (all prefixed with '\')

- d - weekday in “weekay month data” format
- e - ASCII escape character (useful for other control sequences)
- h - hostname (H for FQDN)
- @ - current time in 12-hour AM/PM format
- u - current user
- W - current directory (W for full path)
- # - command number
- $ - $ if normal user, # if root
- nnn - special character mapped to octal number nnn

Questions?

References

Bash

http://mywiki.wooledge.org/BashFAQ
http://mywiki.wooledge.org/BashGuide
http://mywiki.wooledge.org/BashPitfalls
https://tldp.org/HOWTO/Bash-Prog-Into-HOWTO.html

Linux

http://tldp.org
https://linux.org

LaTeX

http://latex.org
https://latex-project.org

Slides And Code

https://github.com/cwpitts/bash-presentation