

Bash Shell Scripting

Benjamin Brewster

Except as noted, all images copyrighted with Creative Commons licenses,
with attributions given whenever available

Shell Scripting

- All of the commands that are accessible from the shell can be placed into a shell “script”
- Shell scripts are executed line by line, as if they were being typed in line by line

Shell Script – High Level

- High level programming language
 - Variables, conditionals, loops, etc.
- Interpreted
 - No compiling, no variable declaration, no memory management
- Powerful/efficient
 - Do lots with little code
 - Can interface with any UNIX program that uses stdin, stdout, stderr
- String and file oriented

When to Use Shell Scripts?

- Automating frequent UNIX tasks
- Simplify complex commands
- For small programs that:
 - ... you need to create quickly
 - ... will change frequently
 - ... need to be very portable
 - ... you want others to see and understand easily
- As a glue to connect together other programs

Hello World!

Note special “shebang” magic characters #!

The first line tells UNIX the path to the shell with which to interpret the script

```
#!/bin/bash
```

Lines beginning with # are comments

```
# Obligatory programming example
```

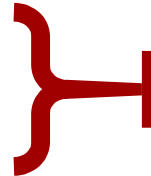
```
echo Hello World!
```

Lines to execute begin here

Variables

- Create a variable by simply assigning to it:

```
myint=1  
mystr=Ben  
mystr=123
```



Note: (very) weakly typed!

- Get the value back out by using the \$ operator in front of the variable:

```
myint=1  
echo $myint
```

Quote Marks – Protecting Your Text

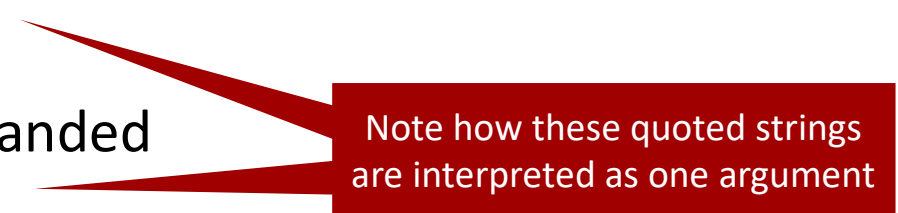
- Quotation marks allow you to control the expansion of variables within strings of text, and group text into a single argument.

- Single Quotes (' '): No variable expansion

```
$ printf 'all your base $abtu\n'  
all your base $abtu
```

- Double Quotes (" "): Variables are expanded

```
$ printf "all your base $abtu\n"  
all your base are belong to us
```



Note how these quoted strings are interpreted as one argument

- Backslash character (\) means evaluate literally, instead of interpret:

```
$ nt="NOTTACHANCE"  
$ printf "\$nt"; printf "$nt\n"  
$ntNOTTACHANCE
```

Printing Example - Demonstration

```
$ echo -e "cat\ndoge\nkat\ndoug" | sort
```

```
cat  
doge  
doug  
kat
```




Somewhat complicated

Versus:

```
$ cAd
```

```
cat  
doge  
doug  
kat
```



Easier to type! How do we
built this script?

Printing Example - Built From Scratch

```
$ echo "#!/bin/bash"
```

```
-bash: !/bin/bash": event not found
```

This occurs because the history feature engages when ! is the first character

```
$ echo "#\x21/bin/bash"
```

```
#\x21/bin/bash
```

Without the -e argument, the hexadecimal ASCII code (21) for ! won't be expanded

```
$ echo -e "#\x21/bin/bash" > cAd
```

```
$ echo -e "echo \"cat\"\\necho \"doge\"\\necho \"kat\"\\necho \"doug\"\" | sort >> cAd
```

```
$ chmod +x cAd
```

```
$ cAd
```

```
cat
doge
doug
kat
```

Make this file be executable
- more on `chmod` later

Editors and Word Processors


- Instead of using “echo >>” to build up a file, we typically use word processing programs like:
 - emacs
 - vi
 - pico
- You can write UNIX files on other operating systems, but the differences in line endings in file formats is a challenge
 - `dos2unix` converts windows line endings to those used by UNIX



Upcoming lecture on vi

Shell Keywords

- Keywords are commands in bash that are interpreted by the shell as delimiters, branching constructs, and loops, among others.
- `compgen` generates possible completion matches for various categories of keywords.
 - `-k` is all reserved keywords
- Can't use keywords as variable names



```
$ compgen -k
if
then
else
elif
fi
case
esac
for
select
while
until
do
done
in
function
time
{
}
!
[[
]]
coproc
```

Environment Variables

- A set of variables that are always available in your shell
- These environment variables control options that change the operation of the shell
- Here are a few common ones:
 - PATH - the set of directories bash will search through to find a command
 - HOME - a shortcut back to your home directory (equivalent to ~)
 - SHELL - the full path to the default shell
 - HOSTNAME - the name of the computer you're currently using

There's about a billion more of these: you can get a complete list of all of them plus their current contents like so:

```
$ printenv
```

Environment Variables - PATH and HOME

```
$ echo $PATH
```

```
/bin:/sbin:/usr/local/bin:/usr/bin:/usr/local/apps/bin:/usr/bin/X11  
:/nfs/stak/faculty/b/brewsteb/bin:.
```

```
$ echo HOME
```

```
HOME
```

Oops

```
$ echo $HOME
```

```
/nfs/stak/faculty/b/brewsteb
```

```
$ echo ~
```

```
/nfs/stak/faculty/b/brewsteb
```

```
$ echo $~
```

```
$~
```

~ is not a variable, it controls the
bash feature called *tilde expansion*

Special Parameters

- A set of variables that are always available in a script
- Here are a few common ones:
 - \$ - the process ID of the script itself (every running process has a unique PID)
 - ? - the return value of the previously terminated command or script
 - # - the number of arguments (positional parameters) given when a script is executed
 - 1 - the first argument (positional parameter) as of when the script was ran
 - 2 - the second argument (positional parameter) as of when the script was ran
 - 3, 4, etc.

These and more are all explained on the page: Built-In Shell Variables

Return Values Examined

- The `exit` bash shell command and C `exit()` function return their results to the `?` variable.
- 0 is interpreted to mean no errors (normal execution), anything else is a result with specific meaning (usually an error)

```
$ cat exittest
#!/bin/bash
ls
printf "result of ls: $?\n"
exit 5

$ exittest
exittest      testloop      prog1
result of ls: 0

$ echo $?
5
```

Return Values Examined

- The `exit` bash shell command and C `exit()` function return their results to the `?` variable.
- 0 means no errors (normal execution), anything else is a result with specific meaning (usually an error)

```
$ echo hello world
```

```
hello world
```

```
$ echo $?
```

```
0
```

```
$ cd ^=^2^21
```

```
=^2^21: No such file or directory.
```

```
$ echo $?
```

```
1
```


The bash `if` Statement - Return Value Enabled

```
if command-list
```

If *command-list* returns 0 ...

```
then
```

```
    command-list
```

```
elif command-list
```

Else If *command-list* returns 0 ...

```
then
```

```
    command-list
```

```
else
```

Otherwise...

```
    command-list
```

```
fi
```

Close the if statement

Error Handling

- The shell will keep executing even if commands have the wrong syntax, delete files, break things, and in general cause havoc
- If you want the shell to exit if any commands have a problem (i.e. return a non-zero value after executing, with some exceptions):
 - Use `-e` with `/bin/bash`
- Most *signals* will kill script immediately
 - E.g.: CTRL-C sends SIGINT

Error Handling - Cry Havoc!

```
$ cat havoc
```

```
#!/bin/bash
```

```
cd qwcein
```

```
ceiqim
```

```
eqo eo
```

```
$ chmod +x havoc
```

```
$ havoc
```

```
./havoc: line 2: cd: qwcein: No such file or directory
```

```
./havoc: line 3: ceiqim: command not found
```

```
./havoc: line 4: eqo: command not found
```




Protecting Your Thesis - Disastrous results

```
#!/bin/bash  
cp thesis.docx thesis_current.docx  
rm -f thesis.docx
```



Protecting Your Thesis - One Fix

```
#!/bin/bash -e  
cp thesis.docx thesis_current.docx  
rm -f thesis.doc
```



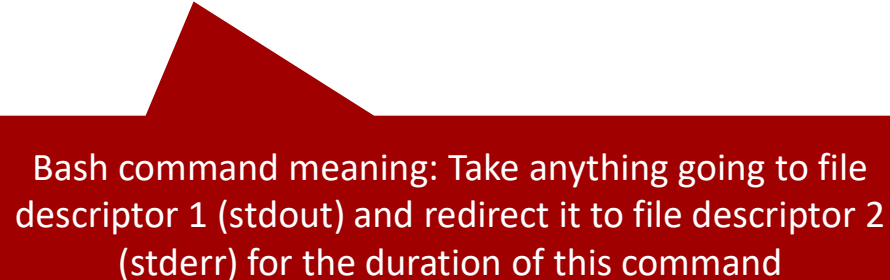
Will only be executed if the result of
the previous command was 0

Protecting Your Thesis - Another Fix

```
#!/bin/bash
if cp thesis.docx thesis_current.docx
then
    rm -f thesis.docx
else
    echo "copy failed" 1>&2
    exit 1
fi
```

Protecting Your Thesis - Another Fix

```
#!/bin/bash
if cp thesis.docx thesis_current.docx
then
    rm -f thesis.docx
else
    echo "copy failed" 1>&2
    exit 1
fi
```



Bash command meaning: Take anything going to file descriptor 1 (stdout) and redirect it to file descriptor 2 (stderr) for the duration of this command

Protecting Your Thesis - Yet Another Fix with `test`

- You can test for file existence, equality of strings, length, permissions, number equality, etc.

Integer not equal option

```
$ test 1 -ne 2
```

```
$ echo $?
```

```
0
```

Value of 0 shows that the outcome of the test is TRUE: 1 does not equal 2

```
$ test 1 -ne 1
```


```
$ echo $?
```

```
1
```

Value of 1 shows that the outcome of the test is FALSE: 1 actually does equals 1

Protecting Your Thesis - Yet Another Fix with `test`

```
#!/bin/bash
cp thesis.docx thesis_current.docx
if test $? -ne 0
then
    echo "copy failed" 1>&2
    exit 1
fi
rm -f thesis.docx
```



Again: bash command meaning: Take anything going to file descriptor 1 (stdout) and redirect it to file descriptor 2 (stderr) for the duration of this command

for Loop Syntax

```
$ cat forloop
```

```
#!/bin/bash
```

```
for i in a b c d
```

```
do
```

```
    printf "<%s>" $i
```

```
done
```

```
printf "\n"
```

```
$ forloop
```

```
<a><b><c><d>
```

```
$
```

while Loop Syntax

Test can also be written as square brackets:
`while [$i -ne 2]`

```
$ cat whileloop
#!/bin/bash
i=0
while test $i -ne 2
do
    printf "i = $i, not stopping yet\n"
    i=$(expr $i + 1)
done
printf "Stopping, i = $i\n"

$ whileloop
i = 0, not stopping yet
i = 1, not stopping yet
Stopping, i = 2
$
```

Huh?

Subshells

- Some operators (like =) and commands require strings or numbers to operate on, not **other commands**:

```
$ cat sumtest
```

```
#!/bin/bash
```

```
num1=9
```

```
num2=3 + $num1
```

```
echo "num2: $num2"
```

```
$ sumtest
```

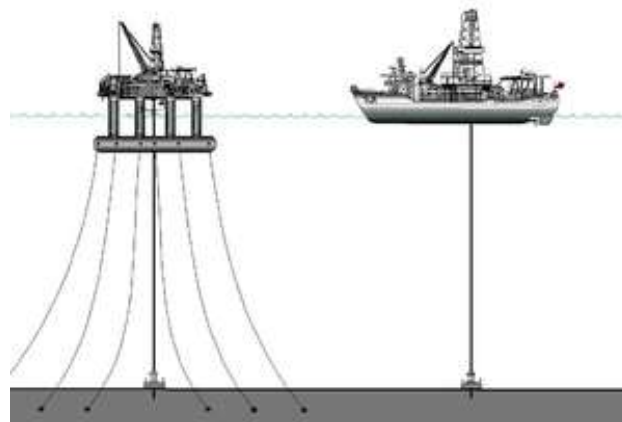
```
./sumtest: line 3: +: command not found
```

```
num2:
```

```
$
```

Subshells

- If we want to increment, or set to an arbitrary value, we first have to calculate that value and return it *as text* to the script, then the script can continue.
- These are executed as entirely separate shells (called subshells) in their own processes that run and return.



Subshells

- Two ways to do command substitution with subshells:

```
i=`expr $i + 1`
```

Backticks! Original method

```
i=$((expr $i + 1))
```

Preferred method: POSIX compliant,
doesn't need escaping when nested

- Both methods evaluate the expression and grab the results from stdout of the subshell
- Double parenthesis construct does arithmetic expansion and evaluation directly, no `expr` command needed:

```
i=$(( 9 + 9 ))
```

```
(( i++ ))
```

Common UNIX Commands - Trapping Signals

- Use the **trap** command to catch signals (like SIGINT generated by hitting CTRL+C) and clean up yer mess.

- Usage:

```
trap <code to execute> list of signals
```



Lots more about
these later

- Example:

```
#!/bin/bash
TMP="myCoolFilename$$"
trap "rm -f $TMP; echo 'CTRL+C received, exiting'; exit 1" INT HUP TERM
echo "lotsa text" > TMP
while [ 1 -ne 2 ]
do
    echo "Never ending loop - hit CTRL+C to exit!"
done
```

Common UNIX Commands - Trapping Signals

- Use the **trap** command to catch signals (like SIGINT generated by hitting CTRL+C) and clean up yer mess.

- Usage:

```
trap <code to execute> list of signals
```

Lots more about
these later

- Example:

```
#!/bin/bash
TMP="myCoolFilename$$"
trap "rm -f $TMP; echo 'CTRL+C received, exiting'; exit 1" INT HUP TERM
echo "lotsa text" > TMP
while [ 1 -ne 2 ]
do
    echo "Never ending loop - hit CTRL+C to exit!"
done
```

Error!

Common UNIX Commands - Trapping Signals

- Use the **trap** command to catch signals (like SIGINT generated by hitting CTRL+C) and clean up yer mess.

- Usage:

```
trap <code to execute> list of signals
```

Lots more about
these later

- Example:

```
#!/bin/bash
TMP="myCoolFilename$$"
trap "rm -f $TMP; echo 'CTRL+C received, exiting'; exit 1" INT HUP TERM
echo "lotsa text" > TMP
while [ 1 -ne 2 ]
do
    echo "Never ending loop"
done
```

Error! Should be \$TMP, otherwise the echo goes to a file called "TMP", instead of a file with name stored in the TMP variable initialized above!