GNU Bash

http://talk.jpnc.info/bash_oscon_2014.pdf

An Introduction to Advanced Usage

James Pannacciulli
Systems Engineer @ (mt) Media Temple

Notes about the presentation:

- This talk assumes you are familiar with basic command line concepts.
- This talk covers **Bash**, not the wealth of CLI utilities available on **GNU/Linux** and other systems.
- This talk assumes a **GNU/Linux** machine, though most everything here should be fairly portable.
- This talk is mostly compatible with **Bash 3**, I'll try to note any examples which require **Bash 4**.
- Bash is fantastic, enjoy the time you spend with it!

Command Types

File:

External executable file.

Builtin:

Command compiled in as part of Bash.

Keyword:

Reserved syntactic word.

Function:

User definable, named compound command.

Alias:

User definable, simple command substitution.

Getting Help

type:

Determine type of command, list contents of aliases and functions.

help:

Display usage information about Bash builtins and keywords.

apropos:

Search man pages.

man:

System manual.

info:

Advanced manual system primarily used for GNU programs.

General reference commands worth running:

man bash

man man

man -a intro

help

help help

info info

info

Some Useful Definitions

word Sequence of **characters** considered to be a single unit.

list Sequence of one or more **commands** or **pipelines**.

name A **word** consisting only of alphanumeric characters and underscores. Can not begin with a numeric character.

parameter An **entity** that stores **values**. A *variable* is a parameter denoted by a *name*; there are also *positional* and *special* parameters.

Return Status

Success:

Command should return a status of **0**.

Failure:

Command should return a **non-zero** status.

- Return values can range from **0** to **255**.
- The return value of the last command to have executed is captured in the special parameter **$?**.
- Many programs signal different types of failure with different return values.

Compound Commands

Iteration:

Continuously loop over **list** of commands delineated by the keywords **do** and **done**.

while until for select

Conditionals:

Execute **list** of commands only if certain conditions are met.

if case

Command groups:

Grouped **list** of commands, sharing any external redirections and whose return value is that of the **list**.

(list) { list; }

While and Until Loops

(Typically) iterate based on an external resource

while list1; do list2; done

Execute **list1**; if **success**, execute **list2** and repeat.
Continue until **list1** returns a **non-zero** status (*fails*).

until list1; do list2; done

Execute **list1**; if **failure**, execute **list2** and repeat.
Continue until **list1** returns a status of **0** (*succeeds*).

The following construct is incredibly handy for processing lines of text: **while read**

For and Select Loops

Iterate based on command line arguments

for name in words; do list; done

During each iteration, assign **name** the value of the next **word**, then execute **list**. Repeat until all **words** have been exhausted.

initialization *condition* *afterthought*

for ((expr1 ; expr2 ; expr3)); do list; done

Evaluate **expr1**, then loop over **list** of commands until **expr2** returns **non-zero** status (*fails*). After each iteration, evaluate **expr3**. The expressions are evaluated as *arithmetic expressions*.

select name in words; do list; done

Create a menu item for each **word**. Each time the user makes a selection from the menu, **name** is assigned the value of the selected **word** and **REPLY** is assigned the **index** number of the selection.

Tests

[expression] or test expression

Evaluate conditional expression with the **test** builtin.

[[expression]]

Evaluate conditional expression with the **[[** keyword; word splitting is **not** performed. The righthand side of a string comparison (**==**, **!=**) is treated as a **pattern when *not* quoted**, and as a **string when quoted**.

[[-n string **]]**	string is **non-empty**
[[-z string **]]**	string is **empty**
[[string1 **==** string2 **]]**	string1 and string2 are the **same**
[[string1 **!=** string2 **]]**	string1 and string2 are **not the same**
[[string **=~** regex **]]**	string **matches** regular expression
[[-e file **]]**	file **exists**
[[-f file **]]**	file is a **regular file**
[[-d file **]]**	file is a **directory**
[[-t fd **]]**	fd is **open** and refers to a **terminal**

Conditionals: if

if list1; then list2; fi

Evaluate **list1**, then evaluate **list2** only if **list1** returns a status of **0**.

if list1; then list2; else list3; fi

Evaluate **list1**, then evaluate **list2** only if **list1** returns a status of **0**. Otherwise, evaluate **list3**.

if list1; then list2; elif list3; then list4; else list5; fi

Evaluate **list1**, then evaluate **list2** only if **list1** returns a status of **0**. Otherwise, evaluate **list3**, then evaluate **list4** only if **list3** returns a status of **0**. Otherwise, evaluate **list5**.

Pattern Matching

Pattern matching is used in Bash for the **[[** *and* **case** *keywords,* **pathname expansion**, *and some types of* **parameter expansion**.

*****	Matches any string, including null.
?	Matches any single character.
[character class]	Matches any one of the characters enclosed between **[** and **]**.

[^...] matches the complement (any character not in the class)

[x-z] matches the range of characters from **x** to **z**

[[:*class***:]]** matches according to these POSIX classes:

alnum alpha ascii blank cntrl digit graph lower print punct space

Conditionals: case

```
case word in
    pattern1)
        list1;;
    pattern2 | pattern3)
        list2;;
esac
```

Match **word** against each **pattern** sequentially. When the first match is found, evaluate the **list** corresponding to that match and stop matching.

The | (pipe) character between two patterns entails a match if either pattern matches (OR).

Command Groups

Subshell:

Evaluate **list** of commands in a **subshell**, meaning that its environment is distinct from the current shell and its parameters are contained.

(list)

Group command:

Evaluate **list** of commands in the **current shell**, sharing the current shell's environment.

{ list ; }

The **spaces** and **trailing semicolon** are *obligatory*.

Redirection

Controlling the input, output, error, and other streams

list > file Overwrite/create **file** with **output** from **list**

list >> file Append/create **file** with **output** from **list**

list < file Feed **file** to **list** as **input**

list1 | list2 Use **output** from **list1** as **input** to **list2**

- If not specified, **fd 1** (**STDOUT**) is assumed when redirecting output.
- Alternative file descriptors may be specified by **prepending** the **fd number**, e.g. **2> file** to redirect **fd 2** (**STDERR**).
- To redirect *to* a file descriptor, append '**&**' and the fd number, e.g. **2>&1** to redirect **STDERR** to the *current target* for **STDOUT**.

Command and Process Substitution

Command substitution:

Replace the **command substitution** **in-line** with the **output** of its **subshell**.

$(*list***)**

Process substitution:

Replace the **process substitution** with a **file descriptor** which is connected to the input or output of the **subshell**.

>(*list***)** **<(***list***)**

Parameters

Positional Parameters: **$1 $2 $3 $4 $5 $6 $7 $8 $9 ${10} ...**

Parameters passed to command, encapsulating **words** on the command line as **arguments**.

Special Parameters: **$* $@ $# $- $$ $0 $! $? $_**

Parameters providing **information** about positional parameters, the current shell, and the previous command.

Variables: name=*string*

Parameters which may be **assigned values** by the user. There are also some special shell variables which may provide information, toggle shell options, or configure certain features.

For variable assignment, "**=**" must **not** have adjacent spaces.

Parameter Expansion: Conditionals

(check if variable is unset, empty, or non-empty)

	unset param	param=""	param="**gnu**"
${param-*default*}	default	–	gnu
${param=*default*}	*name=default*	–	gnu
${param+*alternate*}	–	alternate	alternate
${param?*error*}	error	–	gnu
Treat empty as unset:			
${param:-*default*}	default	default	gnu
${param:=*default*}	*name=default*	*name=default*	gnu
${param:+*alternate*}	–	–	alternate
${param:?*error*}	error	error	gnu

Parameter Expansion: Substrings

param="**racecar**"

Extraction:

offset of **3**, length of **2**

${param:*offset*}

ecar

${param:*offset*:*length*}

ec

Removal from left edge:

pattern is '***c**'

${param#*pattern*}

ecar

${param##*pattern*}

ar

Removal from right edge:

pattern is '**c***'

${param%*pattern*}

race

${param%%*pattern*}

ra

Parameter Expansion: Indirection, Listing, and Length

param="**parade**"; parade="**long**";
name=(**gnu not unix**)

Indirect expansion:

${!param} long

List names matching prefix "pa":

${!pa*} or "${!pa@}" parade param

List keys in array:

${!name[*]} or "${!name[@]}" 0 1 2

Expand to length:

${#param} 6

Parameter Expansion: Pattern Substitution

param="**racecar**"

Substitution: pattern is '**c?**', string is '**T**'

${param/*pattern*/*string*} raTcar

${param//*pattern*/*string*} raTTr

Substitute at left edge: pattern is '**r**', string is '**T**'

${param/#*pattern*/*string*} Tacecar

Substitute at right edge:

${param/%*pattern*/*string*} racecaT

Indexed Arrays

Assign an array by elements:

```
array=( zero one two "three and more" )
```

Add an element to an array:

```
array+=( "four and beyond" )
```

Recreate array with spaces in elements as underscores:

```
array=( "${array[@]// /_}" )
```

Recreate array only with elements from index 2 to 4:

```
array=( "${array[@]:2:3}" )
```

Print element at index 1 of array:

```
echo "${array[1]}"
```

Print array indexes:

```
echo ${!array[@]}
```

Associative arrays are available in Bash 4 and greater.

Arithmetic Expansion

((math and stuff))

name++ increment name after evaluation
name-- decrement name after evaluation

++name increment name before evaluation
--name decrement name before evaluation

- + * / % ** <= >= < > == != && ||

- Can be used as a test, returning 0 if the comparison, equality, or inequality is true, or if the calculated number is not zero.
- Can provide in-line results when used like command substitution – **$((** *math* **))**.
- Bash does not natively support floating point.

Brace Expansion

Arbitrary Word Generation

String generation:

prefix{*ab*,*cd*,*ef*}suffix

Sequence generation:

prefix{*x*..*y*}suffix

Sequencing by specified increment:

prefix{*x*..*y*..*incr*}suffix

Brace expansion may be **nested** and **combined**.

The **prefix** and **suffix** are optional.

Functions

Functions are compound commands which are defined in the current shell and given a function name, which can be called like other commands.

func.name () compound_cmd

Assign **compound_cmd** as function named **func.name**.

func.name () compound_cmd [>,<,>>] file

Assign **compound_cmd** as function named **func.name**; function will always redirect to (**>**), from (**<**), or append to (**>>**) the specified file. Multiple file descriptors may be specified, for instance: ***>out.file 2>err.log***.

Function examples

```
words ()
# print each word on new line
for word
do
  echo "$word"
done
```

Example usages:
words one two 'three four'
words “${BASH_VERSINFO[@]}”

Negative indexing in strings and arrays requires Bash > 4.2. For older versions, the math must be done manually:
${var:$((${#var} - 1))}

```
rev_chars ()
# reverse characters by word
for charlist
do local word
  while (( ${#charlist} ))
  do
    echo -n "${charlist:(-1)}"
    charlist="${charlist:0:(-1)}"
  done
  (( ++word == ${#@} )) &&\
    echo ||\
    echo -n "${IFS:0:1}"
done
```

Example usage:
rev_chars one two 'three four'

Function examples

```
memtop ()
# list top consumers of memory on the system
{
 {
  echo "_PID_ _Name_ _Mem_"
  for i in /proc/[0-9]*
    do
      echo -e "${i##*/}\t$(<$i/comm)\t$(pmap -d "${i##*/}" |\
        tail -1 | {
          read a b c mem d
          echo $mem
        }
      )"
    done |\
    sort -nr -k3 |\
    head -$((${LINES:-23} - 3))
 } |\
 column -t
} 2>/dev/null
```

Example usages:
memtop
export -f memtop; watch bash -c memtop

Session Portability

Import elements from current session directly into a new local or remote session.

```
sudo bash -c “
$(declare -p parameters;
  declare -f functions)
code and stuff”
```

Import **parameters** and **functions** into **root** shell, then run **code and stuff**.

```
ssh remote_host “
$(declare -p parameters;
  declare -f functions)
code and stuff”
```

Import **parameters** and **functions** into **remote** shell, then run **code and stuff**.

- **declare** can list parameters and functions from the current shell, or can set parameter attributes.
- When **sourcing** or **interpolating** Bash code, be mindful of shell options which affect parsing, such as *extglob*, if the code relies on that syntax.

Example code from the talk

```
while read var1 var2; do echo $var2 $var1; done
echo -e 'one two\none two three' > testfile
while read var1 var2; do echo $var2 $var1; done < testfile

for i in one two 'three four'; do echo "_-_-_-$i-_-_-_"; done

select choice in one two 'three four'; do echo "$REPLY : $choice"; done

if [ "a" == "a" ]; then echo "yep"; else echo "nope"; fi
if [ "a" == "b" ]; then echo "yep"; else echo "nope"; fi

case one in o) echo 'o';; o*) echo 'o*';; *) echo 'nope';; esac

unset x
(x=hello; echo $x); echo $x
{ x=hello; echo $x; }; echo $x

echo b; echo a | sort
(echo b; echo a) | sort
```

Example code from the talk

```
echo "$(echo "$(echo "$(echo "$(ps wwf -s $$)")")")"
echo this `echo quickly \`echo gets \\\`echo very \\\\\\\`echo ridiculous\\\\\\\`\\\`\``
echo "$(</etc/os-release)"

PS1="[$?] $PS1"   # show exit status of prev. cmd in prompt
[ -t 0 ]
[ -t 0 ] < /etc/os-release

testvar="hello world"
[ $testvar == "hello world" ]  # fails
[ "$testvar" == "hello world" ]
[[ $testvar == "hello world" ]]
[[ $testvar == hello?w*d ]]

(( 0 ))
(( 1 ))
echo $(( 3 * 2 - (11 * 5) ))
```

Example code from the talk

```
echo bash{,e{d,s},ful{,ly,ness},ing}
echo {1..5}{0,5}%
echo {10..55..5}%
echo {a..z..12}
man{,}
cp -v filename{,.bak}   # quick backup of filename

sudo bash -c “$(declare -f words); words one two 'three four'”

ssh localhost “$(declare -f memtop); memtop”
```

On an unrelated note, Bash can actually complete (like tab completion) a list of files into nested brace expansion format with the **ESC-{** key combination. All key bindings may be displayed with **bind -P**.

A Few Good Links

- http://www.gnu.org/software/bash/
- http://tiswww.case.edu/php/chet/bash/NEWS
- http://tldp.org/LDP/abs/html/index.html
- http://wiki.bash-hackers.org/doku.php