



Concise!

GNU Bash

http://talk.jpnc.info/bash_linuxcon-eu.pdf

an introduction to advanced usage

James Pannacciulli

Systems Engineer in Web Hosting

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.

```
command types
[0] ~/bash$ type -a \
> ls cd while genpass
ls is aliased to `ls --color=auto'
ls is /bin/ls
cd is a shell builtin
while is a shell keyword
genpass is a function
genpass ()
{
    tr -dc 'a-zA-Z0-9_#@.-' < /dev/urandom | head -c ${1:-14};
    echo
}
[0] ~/bash$
```

Getting Help with Bash and with your OS

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

help

info

man man

help help

man -a intro

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.

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**.

Tests

[**expression**] or **test expression**

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

[[**expression**]]

Evaluate **conditional expression** with the [[keyword; word splitting is **not** performed during any parameter expansion. 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 == pattern]] | string matches pattern |
| [[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 |

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**).

Parameters

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

Parameters passed to salient 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

Extraction:

`${param:offset}`

`${param:offset:length}`

Removal from left edge:

`${param#pattern}`

`${param##pattern}`

Removal from right edge:

`${param%pattern}`

`${param%%pattern}`

`param="racecar"`

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

ecar

ec

pattern is '***c**'

ecar

ar

pattern is '**c***'

race

ra

Parameter Expansion: Pattern Substitution

Substitution:

`${param/pattern/string}`

`${param//pattern/string}`

Substitute at left edge:

`${param/#pattern/string}`

Substitute at right edge:

`${param/%pattern/string}`

param="racecar"

pattern is 'c?', string is 'T'

raTcar

raTTT

pattern is 'r', string is 'T'

Tacecar

racecaT

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

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 (second element):

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

Print array indexes:

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

Associative arrays are available in Bash 4 and greater.

Arithmetic Expressions

((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 expansion 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 (Bash 4+):

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

Bash can complete a list of files into nested brace expansion format with the **ESC-`{`** key combination. All key bindings may be displayed with **bind -P**.

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

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

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 **expr2 && { list; expr3; }** – that is to say execute **list** until **expr2** returns **non-zero** status (*fails*), evaluating **expr3** after each iteration. The expressions are evaluated as *arithmetic expressions*, and the **list** as a regular command list.

select name in words; do list; done

Create a menu with each **word** as an item. When the user makes a selection, **name** is assigned the value of the selected **word**, **REPLY** is assigned the **index** number of the selection, and **list** is executed.

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.

The righthand side of a pipe is *always* a subshell. **(list)**

Group command:

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

{ 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 a file.
- To redirect *to* a file descriptor, append **'&'** and the **fd number**, e.g. **2 > &1** to redirect **STDERR** to the *current target during parsing* for **STDOUT**.

Command and Process Substitution

Command substitution:

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

`$(list)`

Process substitution:

Replace the **process substitution** with a **file descriptor** which is connected to the input or output of the **subshell**. Allows *commands* in **list** to act as a *file*.

`>(list) <(list)`

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** to function named **func.name**.

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

Assign **compound_cmd** to 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**.

Session Portability

Import elements from current session 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

```
true  
echo $?
```

```
false  
echo $?
```

```
if fgrep -qi gentoo /etc/os-release  
then  
    echo "gentoo"  
else  
    echo "not gentoo"  
fi
```

```
if fgrep -qi arch /etc/os-release  
then  
    echo "arch"  
else  
    echo "not arch"  
fi
```

Example code from the talk

```
[[ -n "much content!" ]]
```

```
[[ -z "wow!" ]]
```

```
[[ -e /etc ]] && echo exists
```

```
[[ -f /etc ]] && echo regular file
```

```
[[ -d /etc ]] && echo directory
```

```
[[ -t 0 ]]
```

```
[[ -t 0 ]] < /etc/os-release
```

```
if [[ "abc" == "abc" ]]
```

```
then
```

```
    echo "yep"
```

```
else
```

```
    echo "nope"
```

```
fi
```

Example code from the talk

```
if [[ "abc" == "c" ]]
then
  echo "yep"
else
  echo "nope"
fi
```

```
if [[ "abc" == *c ]]
then
  echo "yep"
else
  echo "nope"
fi
```

```
[[ "linuxcon europe" == [a-z]*[^:digit:] ]]
```

```
[[ "linuxcon europe" == *[^d-h] ]]
```

Example code from the talk

```
case one in
  o)
    echo 'o'
  ;;
  o?e)
    echo 'o?e'
  ;;
  o*)
    echo 'o*'
  ;;
  *)
    echo 'nope'
  ;;
esac
```

```
set -- one two "three four" five
printf "%s\n" "\$1: $1" "\$2: $2" "\$3: $3" "\$4: $4" "\$5: $5" "\$#: $#"\
"\$*: $*" "\$@: @$"
```

```
param=gnu; echo "${param:-default value for expansion}"
```

Example code from the talk

```
unset param; echo "${param:-default value for expansion}"
```

```
echo "${param:?a nifty custom error string}"
```

```
echo "${PATH:+yes you have a PATH, great job}"
```

```
echo "${BASH_VERSION:0:1}"
```

```
echo "${PATH##*:*}"
```

```
echo -e "${PATH//:/^\\n}"
```

```
param=PATH; printf "%s\\n\\n" "\\$param: ${param}"\  
"\\${!param}: ${!param}" "\\${!param%%:*}: ${!param%%:*}"
```

```
echo ${!BASH*}
```

```
echo "${#PATH}"
```

```
array=( zero one two "three and more" )  
printf "%s\\n" "${array[@]}"
```

Example code from the talk

```
array+=( "four and beyond" )  
printf "%s\n" "${array[@]}"
```

```
array=( "${array[@]// /_}" )  
printf "%s\n" "${array[@]}"
```

```
array=( "${array[@]:2:3}" )  
printf "%s\n" "${array[@]}"
```

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

```
echo $(( 3 + 11 ))
```

```
(( 3 >= 5 ))
```

```
(( 0 ))
```

```
echo $(( i++ ))
```

```
echo bash{e{d,s},ful{,ly,ness},ing}
```


Example code from the talk

```
echo {1..5}{0,5}%
```

```
echo {10..55..5}%
```

```
echo {a..z..12}
```

```
touch testfile && cp -v testfile{,.bak}
```

```
man{,}
```

```
while read var1 var2  
do  
    echo $var2 $var1  
done
```

```
count=0  
until (( ++count > 3 ))  
do  
    echo $count  
done
```

Example code from the talk

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

```
for (( i=0 ; i<5 ; i++ ))
do
  echo $i
done
```

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

```
for file in *
do
  echo "$(stat -c"%a %A" "$file") $(md5sum "$file")"
done
```

Example code from the talk

```
ls -1 | while read file
do
    echo "$(stat -c"%a %A" "$file") $(md5sum "$file")"
done
```

```
select file in *
do
    stat "$file"
    break
done
```

```
unset x
(x=hello; echo "x: $x")
echo "x: $x"
```

```
unset x
{ x=hello; echo "x: $x"; }
echo "x: $x"
```

```
printf "%s\n" ${RANDOM:1:2} ${RANDOM:1:2} ${RANDOM:1:2} | sort -n
```

Example code from the talk

```
man bash          \
tr [[:space:]] "\n" \
tr A-Z a-z       \
grep -v "^[[:space:]]*$" \
sort             \
uniq -c         \
sort -n        \
tail -${( ${LINES:-16} - 1 )}
```

```
echo b; echo a | sort
```

```
{ echo b; echo a; } | sort
```

```
echo "what a wonderful example" > awesome.txt
cat < awesome.txt
```

```
filename="file_$(date +%F)"
echo "$(date +%s)" > "$filename"
sleep 1s
echo "$(date +%s)" >> "$filename"
```

Example code from the talk

```
printf "%s\n"\  
"$filename: $(wc -l "$filename" | cut -d" " -f1) lines"\  
""\  
"$(<"$filename")"
```

```
echo "$(echo "$(echo "$(echo "$(ps wwf -s $$)")")")")"  
echo this `echo quickly `echo gets \\`echo very \\\\\\\\\\\`echo  
extremely \\\\\\\\\\\\\\\\\\\\\\\`echo ridiculous\\\\\\\\\\\\\\\\\\\\\\\\\\`\\\\\\\\\\\\\\`\\\\\\`\\`
```

```
wc -c <(echo "$PATH")  
wc -c < <(echo "$PATH")
```

```
printf "%s\n" one two "three four" \  
tee >(tac) >(sleep 1; cat) >/dev/null \  
cat
```

```
var=$(  
printf "%s\n" one two "three four" \  
tee >(tac) >(sleep 1; cat) >/dev/null  
)  
echo "$var"
```

Example code from the talk

```
unset array
while read; do
  array+=( "$REPLY" )
done
declare -p array
```

```
unset array
# WILL NOT WORK
printf "%s\n" one two "three four" | \
  while read; do
    array+=( "$REPLY" )
  done
declare -p array
```

```
unset array
while read; do
  array+=( "$REPLY" )
done <<(printf "%s\n" one two "three four")
declare -p array
```

Example code from the talk

```
diff -wyW85\  
  <(echo "${examples[((I - 2))]}")\  
  <(echo "${examples[((I - 1))]}") |\  
  highlight --syntax bash -O xterm256 -s rootwater
```

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

Example code from the talk

```
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
```

```
rev_words ()
# reverse/print each word on new line
for word
do
  echo "$(rev_chars "$word")"
done
```


Example code from the talk

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

Example code from the talk

```
random_word ()
{
  local word= count=1;
  while ;; do
    word=$(tr -dc 'a-z' < /dev/urandom | head -c ${1:-4})
    fgrep -qi $word /usr/share/dict/cracklib-small && {
      echo $count: $word
      return 0
    } || (( count++ ))
  done
}
```

```
for container in 172.17.0.{1..5}
do
  printf "%s\n" "$container: $(
    ssh -o StrictHostKeyChecking=no -i ~/.ssh/docker.id_rsa $container \
      "$(declare -f random_word); random_word" )"
done
```

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>
- <http://git.jpnc.info/parssh/>