



# GNU Bash

*An Introduction to Advanced Usage*

[http://talk.jpnc.info/bash\\_uuasc\\_2013-06.pdf](http://talk.jpnc.info/bash_uuasc_2013-06.pdf)

James Pannacciulli

Sysadmin @ (mt) Media Temple

# Notes about the presentation:

This is a talk about Bash, not about GNU/Linux in general, and not about the wealth of high quality command line utilities which are often executed from within Bash.

The assumed operating system is GNU/Linux, with a recent version of Bash. This talk is almost entirely Bash 3 compatible; I will try to point out if any features or examples require Bash 4.

I do not consider myself an *expert*. I am a **professional user** and an **enthusiast** and I want to share some of what I am learning, because Bash is a wonderful shell.

# Command Types

## File:

External executable file.

## Keyword:

Reserved syntactic word.

## Builtin:

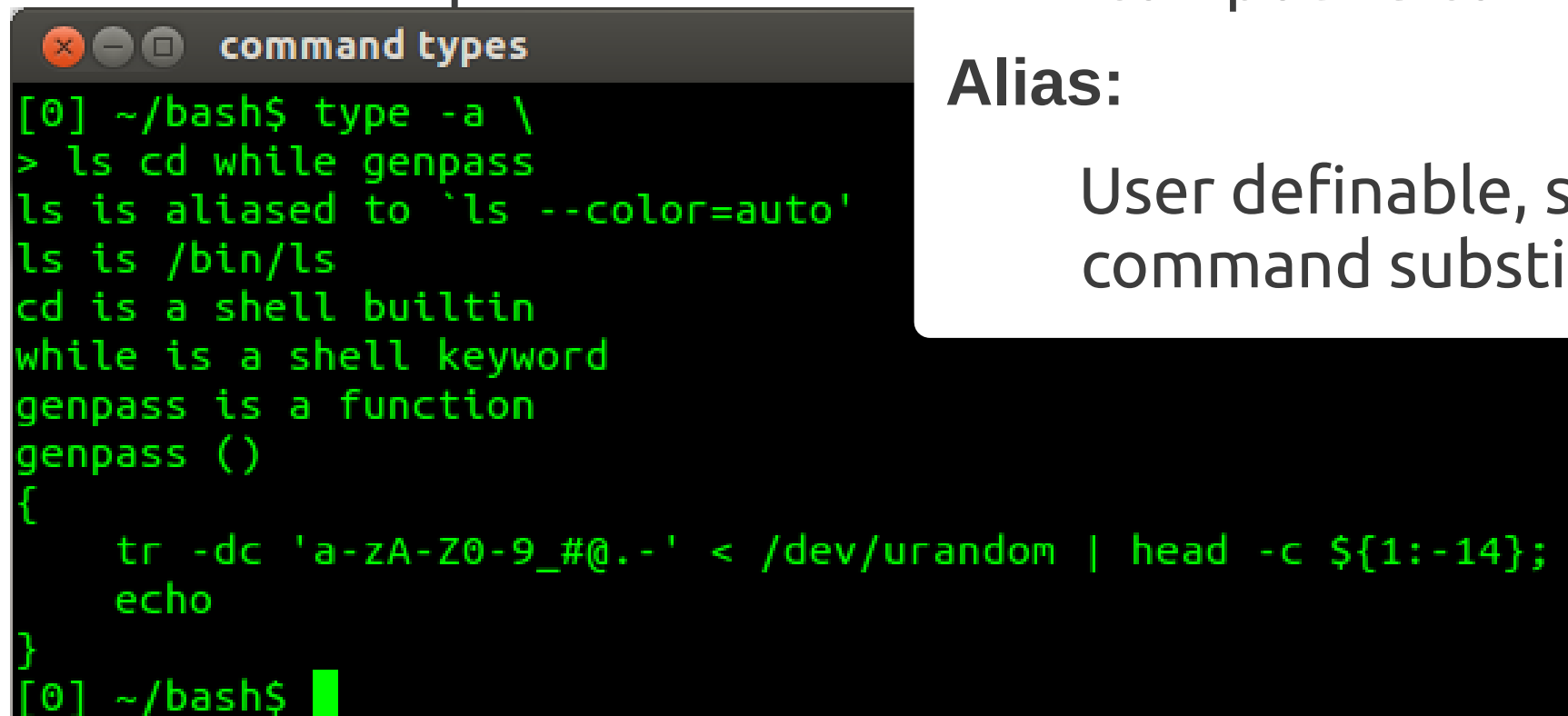
Command compiled in as part of Bash.

## 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

## 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.

# 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

**while list1; do list2; done**

Loop over **list2** of commands until **list1** returns a **non-zero** status (*fails*).

**until list1; do list2; done**

Loop over **list2** of commands until **list1** returns a status of **0** (*succeeds*).

The following construct is incredibly handy for processing lists of items: **while read**



# For and Select Loops

**for** *name* **in** *words*; **do** *list*; **done**

Loop over **list** of commands, assigning **name** the value of each **word** 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 and pathname expansion are **not** performed. The righthand side of a string comparison (**==**, **!=**) is treated as a **pattern** when not quoted, and an additional regular expression operator, **=~**, is available.

**-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**

**-e** file      file **exists**

**-f** file      file exists and is a **regular file**

**-d** file      file exists and 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.

# 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*.

# Command and Process Substitution

## Command substitution:

Replace the **command substitution** with the **output** of its **subshell**.

***\$(list)***

## Process substitution:

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

***>(list)***   ***<(list)***

# Parameters

## Positional Parameters:

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

**\$1 \$2 \$3 \$4 \$5 \$6 \$7 \$8 \$9 \${10} \${11} ...**

## Special Parameters:

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

**\$\* @\$ \$# \$- \$\$ \$0 \$! \$? \$\_**

## Variables:

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.

**name=string**

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: Indirection, Listing, and Length

```
param="parade"; parade="long";  
name=( gnu not unix ); prefix is "pa"
```

Indirect expansion:

`${!param}`

long

List names matching prefix:

`${!prefix*}` or `"${!prefix@}"`

parade param

List keys in array:

`${!name[*]}` or `"${!name[@]}"`

0 1 2

Expand to length:

`${#param}`

6

# 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

raTTTr

pattern is 'r', string is 'T'

Tacecar

racecaT

# 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[@]}
```

# 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**, which will always redirect to (>), from (<), or append to (>>) the specified file. One or more file descriptors may also be specified this way (>&2, for instance).



# 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[@]}"
```

```
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##*/}" |\n  
        tail -1 | {  
          read a b c mem d  
          echo $mem  
        })"  
    done |\n    sort -nr -k3 |\n    head -${((LINES - 3))}  
  } |\n  column -t  
} 2>/dev/null
```

Example usages:

memtop

export -f memtop; watch bash -c memtop

# Example code from the talk

```
while read var1 var2; do echo $var2 $var1; done  
echo -e 'one two\nnone 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 "$ (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
```

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 the **bind -P** command.

# 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>