



*Concise!*  
**GNU Bash**

[http://talk.jpnc.info/bash\\_lfnw\\_2017.pdf](http://talk.jpnc.info/bash_lfnw_2017.pdf)

*an introduction to advanced usage*

James Pannacciulli

Systems Engineer

# 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.
- Bash is flexible and fun, don't forget to enjoy the time you spend using it!

# 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 with Bash and with your OS

## type:

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

## apropos:

Search man pages.

## help:

Display usage information about  
Bash builtins and keywords.

## man:

System manual.

## info:

Advanced manual system  
primarily used for GNU  
programs.

## General reference commands to get started:

man bash

help

info

man man

help help

man -a intro

info info

# Some Useful Definitions

*Technical Terms as Defined and Used in Bash Documentation*

**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 returns a status of **0**.

**Failure:** Command returns a **non-zero** status.

- Valid return values 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 or error with different return values, which allows us to handle errors programmatically.

# List Operators

**list0; list1**

Execute **list0**, then execute **list1**. Same as separation by newline.

**list0 & list1**

Execute **list0** in a background subshell and simultaneously execute **list1**.

**list0 && list1**

Execute **list0**, then execute **list1** only if **list0** returns status 0.

**list0 || list1**

Execute **list0**, then execute **list1** only if **list0** returns a non-zero status.

# Conditionals: if

```
if list0  
  then list1  
fi
```

Evaluate *list0*, then evaluate *list1*  
only if *list0* returns status *0*.

```
if list0  
  then list1  
  else list2  
fi
```

Evaluate *list0*, then evaluate *list1*  
only if *list0* returns status *0*.  
Otherwise, evaluate *list2*.

```
if list0  
  then list1  
  elif list2  
    then list3  
    else list4  
fi
```

Evaluate *list0*, then evaluate *list1*  
only if *list0* returns status *0*.  
Otherwise, evaluate *list2*, then  
evaluate *list3* only if *list2* returns  
status *0*. Otherwise, evaluate *list4*.



# Tests

[ expression ]

test expression

Evaluate **conditional expression** with the **test** builtin (or the analogous **/bin/[** or **/bin/test** commands if specified).

[[ 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**.
- Regular Expressions may be matched with the **=~** operator.
- Short circuiting logical operators **&&** and **||** can be used to combine condition expressions.

# Common Conditional Expressions

See them all by executing **help test**

**[[ -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**

**[[ file0 -nt file1 ]]** file0 is **newer than** file1

**[[ file0 -ef file1 ]]** file0 is a **hard link to** file1

**[[ -n string ]]** string is **non-empty**

**[[ -z string ]]** string is **empty**

**[[ string0 == "string1" ]]** string0 and string1 are the **same**

**[[ string0 != "string1" ]]** string0 and string1 are **not the same**

**[[ string == pattern ]]** string **matches** pattern

**[[ string =~ regex ]]** string **matches** regular expression

# 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  
  pattern0)  
    list0;;  
  pattern1 | pattern2)  
    list1;;  
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 ( *inclusive 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

param="mandrake"

## Extraction:

`${param:offset}`

`${param:offset:length}`

## Removal from left edge:

`${param#pattern}`

`${param##pattern}`

## Removal from right edge:

`${param%pattern}`

`${param%%pattern}`

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

drake

dr

pattern is '**\*a**'

ndrake

ke

pattern is '**a\***'

mandr

m

# 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="ubuntu"

pattern is 'u?', string is 'X'

Xuntu

XXtu

pattern is 'u', string is 'X'

Xbuntu

ubuntX



# Parameter Expansion:

Indirection, Element Listing, and Length

```
name0="name1"; name1="hello";  
array=( gnu not unix )
```

**Indirect expansion:**

`${!name0}`

hello

**List names matching prefix “na”:**

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

name0 name1

**List keys in array:**

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

0 1 2

**Expand to length:**

`${#name0}`

5

`${#array}`

3

# Indexed Arrays

**Assign an array by elements:**

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

**Add to an array or modify an element:**

```
array+=( "four and beyond" [0]=ZERO )
```

**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[@]}
```

# 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 under certain conditions.  
**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 list0; do list1; done**

Execute **list0**; if **success**, execute **list1** and repeat.  
Continue until **list0** returns a **non-zero** status (*fails*).

**until list0; do list1; done**

Execute **list0**; if **failure**, execute **list1** and repeat.  
Continue until **list0** 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 (( expr0 ; expr1 ; expr2 )); do list; done**

Evaluate **expr0**, then loop over **((expr1)) || break; { list; ((expr2)); }** – that is to say execute **list** only if **expr1** returns **non-zero** status (*fails*), evaluating **expr2** 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

*Treat group as single unit for redirection or branching*

## 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*.



# Associative Arrays

Assign an array by elements:

```
declare -A array=(  
    [item]=cheese [price]=6.75 )
```

Add to or modify elements in an array:

```
array+=( [type]="fresh curds" )
```

Copy array:

```
declare -n array2=array
```

List array keys:

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

List array values:

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

Associative arrays  
( `array[key]=value` ) may be  
created in Bash 4 and greater  
with `declare -A array`.

# 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**

**list0 | list1**    Use **output** from **list0** as **input** to **(list1)**

- 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, prepend '&' 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 distinct 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.

# 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/>