bw|HPC – C5

bwHPC Course: (Advanced) Bash Scripting

Robert Barthel

UNIVERSITÄT HEIDELBERG ZUKUNFT SEIT 1386

Hochschule für Technik Stuttgart

UNI FREIBURG

Hochschule Esslingen University of Applied Sciences

Universität Konstanz

UNIVERSITÄT MANNHEIM

Universität Stuttgart

EBERHARD KARLS UNIVERSITÄT TÜBINGEN

KIT Karlsruher Institut für Technologie

ulm university universität uulm

Funding: Baden-Württemberg MINISTERIUM FÜR WISSENSCHAFT, FORSCHUNG UND KUNST

How to read the following slides

Abbreviation/Colour code	Full meaning
`$ command -option value`	`$` = prompt of the interactive shell The full prompt may look like: `user@machine:path$` The command has been entered in the interactive shell session
`<integer>` `<string>`	<> = Placeholder for integer, string etc
`foo, bar`	Metasyntactic variables
`${WORKSHOP}`	`/pfs/data1/software_uc1/bwhpc/kit/workshop/2015-09-22`

Sources of this slides?

- http://tldp.org/HOWTO/Bash-Prog-Intro-HOWTO.html (intro)
- http://tldp.org/LDP/abs/html (advanced)
- $ man bash

Why (not) Bash?!

- Great at:
  - managing batch jobs
  - managing external programs
  - invoking entire UNIX command stack & many builtins
- Powerful scripting language
- Portable and version-stable
- Bash almost everywhere installed

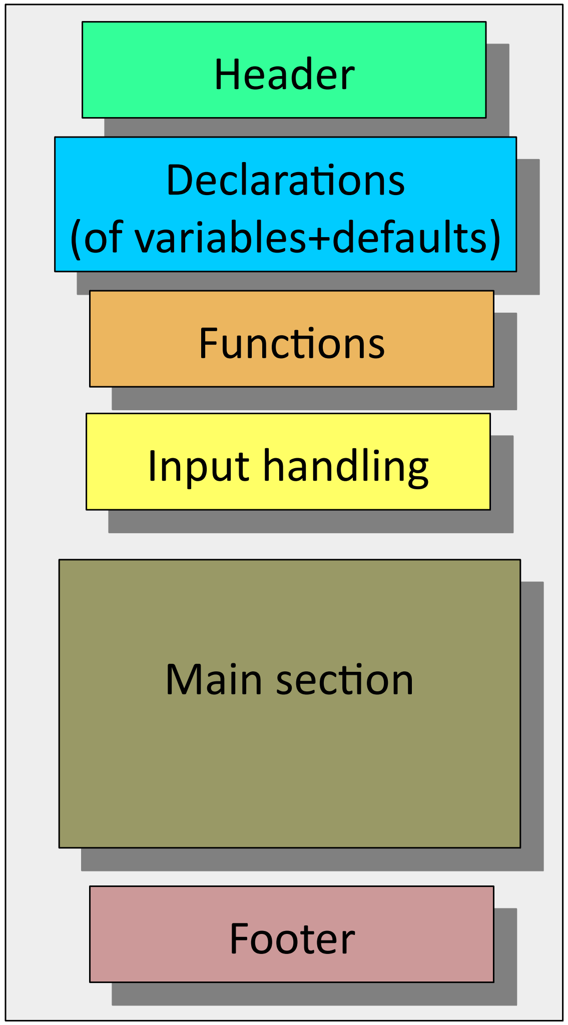
- Less useful when:
  - Resource-intensive tasks (e.g. sorting, recursion, hashing)
  - Heavy-duty math operations
  - Extensive file operations
  - Need for native support of multi-dimensional arrays

Goal

- Be descriptive!
  - Comment your code
    - e.g. via headers sections of script and functions.
  - Decipherable names for variables and functions

- Organise and structure!
  - Break complex scripts into simpler modules

    e.g. use functions
  - Use exit codes
  - Use standardized parameter flags for script invocation.



Header & Line format

- Sha-Bang = '#!'

  → at head of file = 1. line only!

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

  - Options: e.g. debugging shell):

```
#!/bin/bash -x
```

  - #!/bin/sh → invokes default shell interpreter → mostly Bash
  - If path of bash shell varies:

```
#!/usr/bin/env bash
```

- Line ends with no special character!
  - But multiple statements in one line to be separated by:

`;` Comma

```
$ echo hello; echo World; echo bye
```

Bash Output

- **echo**

  a) trails every output with a „newline“

```
$ echo hello; echo World
hello
World
```

  b) prevent newline:

```
$ echo -n hello; echo World
helloWorld
```

  c) parsing „escape sequences“

```
$ echo -e "hello\nWorld"
hello
World
```

- printf = „enhanced“ echo
  - by default no „newline“ trailing
  - formated output

```
$ printf hello; printf World
helloWorld$
```

```
$ printf "%-9.9s: %03d\n" "Integer" "5"
Integer  : 005
```

Globbing

- = filename expansion
  → recognices and expands „wildcards“
- but this is **not** a Regular Expression interpretion

- wildcards:

*	= any multiple characters
?	= any single character
[]	= to list specific character e.g. list all files starting with a or b `$ ls [a,b]*`
^	= to negate the wildcard match e.g. list all files not starting with a `$ ls [^a]*`

Variables (1)

- Substitution:
  - No spaces before and after '='
  - Brace protect your variables!
  - Values can be generated by commands

```
var1=value
```

```
var2=${var2}
```

```
var2=$(date)
```

- Bash variables are untyped

  → essentially strings,

  → depending on content arithmetics permitted

```
$ a=41; echo $((a+1))
42
```

```
$ a=BB; echo $((a+1))
1
```

→ string has an integer value of 0

- declare/typeset (bash builtin)
  - set variable to integer, readonly, array etc.

```
$ declare -r a=1
$ let a+=1
bash: a: readonly variable
```

```
$ array=( '1 2' 3 4 5)
$ echo ${array[0]}
1 2
```

→ space is separator

Variables (2)

- declare – *cont.*
  - Excursion: store file content in array:
    - a) 1 element per string
    - b) 1 element for whole file
    - c) 1 element per line
  - Identifying variable var:

```
a=( $(< file) )
```

```
a=( "$(< file)" )
```

```
while read -r line; do a+=("${line}") ; done < file
```

```
declare | grep var
```

- Usage only without $ prefix when declare, assign, export, unset

```
declare -i a=41
export a
echo ${a}
unset a
echo ${a}
```

→ use **${}** instead of simply $ to avoid problems manipulating variables

Comments and Quotes

Comments

- at beginning
- at the end
- exception: escaping, quotes, substitution

```
# This line is not executed
```

```
echo 'something' # Comment starts here
```

\ Escape = Quoting mechanism for single characters

```
echo \#
```

' Full Quotes = Preserves all special characters within

```
echo '#'
```

" Partial Quotes = Preserves some of the special characters, but not `${var}`

```
var=42

echo "\${var} = ${var}"
echo '\${var} = ${var}'
```

Manipulation of Variables (1)

Syntax	Does?	Examples

`${#var}`	String length	`$ A='abcdef_abcd'; echo ${#A}`
		`11`

`${var:pos:len}`	Substring extraction:	

	a) via Parameterisation	`$ POS=3; echo ${A:${POS}:2}`
		`de`

	b) Indexing from right	`$ echo ${A:(-2)}`
		`cd`

`${var#sstr}`	Strip shortest match of $sstr from front of $var	`$ sstr=a*b; echo ${A#${sstr}}`
		`cdef_abcd`

`${var%sstr}`	Strip shortest match of $sstr from back of $var	`$ sstr=c*d; echo ${A%${sstr}}`
		`abcdef_ab`

`${var/sstr/repl}`	Replace first match of $sstr with $repl	`$ sstr=ab; rp=AB; echo ${A/${sstr}/${rp}}`
		`ABcdef_abcd`

`${var//sstr/repl}`	Replace all matches of $sstr with $repl	`$ echo ${A//${sstr}/$rp}`
		`ABcdef_ABcd`

`${var/#sstr/repl}`	If $sstr matches frond end, replace by $repl	`$ sstr=a; rp=z_; echo ${A/#${sstr}/${rp}}`
		`z_bcdef_abcd`

`${var/%sstr/repl}`	If $sstr matches back end, replace by $repl	`$ sstr=d; rp=_z; echo ${A/%${sstr}/${rp}}`
		`abcdef_abc_z`

Manipulation of Arrays

Syntax	Does?	Examples

`${#array[@]}`	Number of elements	`$ dt=( $(date) ); echo ${#dt[@]}`
		`6`

`${array[@]:p1:p2}`	Print elements from no. **p1** to **p2**:	`$ echo ${dt[@]:1:2}`
		`Feb 25`

`${array[@]#sstr}`	Strip shortest match of $sstr from front of all elements of Array	`$ sstr=W*d; echo ${dt[@]#${sstr}}`
		`Feb 25 10:18:22 CET 2015`

- Adding elements to an array:

  a) at the end:

```
$ dt+=( "AD" )
$ echo ${dt[@]}
Wed Feb 25 17:18:22 CET 2015 AD
```

  b) in-between

```
$ dt=( ${dt[@]:0:2} ':-)' ${dt[@]:2}} )
$ echo ${dt[@]}
Wed Feb 25 :-) 17:18:22 CET 2015
```

Output & Input Redirection (1)

Syntax	Does?	Examples
`exe > log`	Standard output (stdout) of application exe is (over)written to file log	`$ date > log; cat log`
`exe >> log`	Standard output (stdout) of application exe is append to file log	`$ date >> log; cat log`
`exe 2> err`	Standard output (stderr) of application exe is (over)written to file err	`$ date 2> err; cat err`
`exe 2>> log`	Standard output (stderr) of application exe is append to file log	`$ date 2>> err; cat err`
`exe >> log 2>&1`	Redirects stderr to stdout	`$ date >> log 2>&1`

`exe1 | exe2`	Passes stdout of exe1 to standard input (stdin) of exe2 of next command	`# Print stdout & stderr to screen and then append both to file`
		 `$ date 2>&1 | tee -a log`

`exe < inp`	Accept stdin from file inp	`$ wc -l < file`

Output & Input Redirection (2)

- Take care of order when using redirecting
  - e.g:

```
ls -yz >> log 2>&1
```

→ Stdout redirected to file
→ Stderr redirected to file redirectd stdout

```
ls -yz 2>&1 >> log
```

→ Stderr redirected to stdout (channel)
→ Stdout redirected to file

- Suppressing stderr

```
ls -yz >> log 2>/dev/null
```

Usage? Keep variable empty when error occurs,

→ e.g. number of files with extension log

```
list_logs="$(ls *.log 2>/dev/null)"
```

Output & Input Redirection (3)

- Redirection of „all“ output in shell script to one user file

  → generalise = define variable

```
#!/bin/bash

log="blah.log"
err="blah.err"

echo "value 1" >> ${log} 2>> ${err}
command >> ${log} 2>> ${err}
```

  → or use exec

```
#!/bin/bash

exec > "blah.log" 2> "blah.err"

echo "value 1"
command
```

→ all stdout and stderr after 'exec' will be written to blah.log and blah.err resp.

Manipulation of Variables (2)

- Example:

```
${WORKSHOP}/exercises/bash/var_manipulation.sh
```

```
#!/bin/bash

## Purpose: Define automatic output names for executables

exe="binary.x"

## Assume: $exe contains extension .x or .exe etc
sstr=".*"
log="${exe%${sstr}}.log"  ## replace extension with .log
err="${exe%${sstr}}.err"  ## replace extension with .err

## Define command: echo and run
echo "${exe} >> ${log} 2>> ${err}"
${exe} >> ${log} 2>> ${err}
```

Expansion of Variables

Syntax	Does?	Examples

`${var-${def}}`	If $var not set, set value of $def	`$ unset var; def=new; echo ${var-${def}}`
		`new`

`${var:-${def}}`	If $var not set or *is empty*, set value of $def	`$ var=''; def=new; echo ${var:-${def}}`
		`new`
		`# Output name for interactive and MOAB`
		`jobID=${MOAB_JOBID:-${BASHPID}}`

`${var:?${err}}`	If $var not set or *is empty,* print $err and abort script with exit status of 1	`$ var=''; err='ERROR - var not set'`
		`$ echo ${var:?${err}}`
		`bash: var: ERROR - var not set`

Exit & Exit Status

- Exit terminates a script
- Every command returns an exit status
  - successfull = 0
  - non-successfull > 0 (max 255)
- $? = the exit status of last command executed (of a pipe)

```
ls -xy; echo $?
2
```

- Special meanings (avoid in user-specified definitions):

1 =	Catchall for general errors
2 =	Misuse of shell builtins
126 =	Command invoked cannot execute (e.g. /dev/null)
127 =	"command not found"
128 + n =	Fatal error signal "n" (e.g. kill -9 of cmd in shell returns 137)

(Conditional) Tests

```
if condition1 ; then
   do_if_cond1_true/0
elif condition2 ; then
   do_if_cond2_true/0
else
   do_the_default
fi
```

condition	Does?	Examples

`(( ))`	Arithmetic evaluation	`$ if (( 2 > 0 )) ; then echo yes ; fi`
		`yes`

`[ ]`	Part of (file) test builtin, arithmetic evaluation only with `-gt`, `-ge`, `-eq`, `-lt`, `-le`, `-ne`	`$ if [ 2 -gt 0 ] ; then echo yes ; fi`
		`yes`
		`$ # existance of file`
		`$ if [ -e "file" ] ; then echo yes ; fi`

`[[ ]]`	Extented test builtin; allows usage of `&&`, `||`, `<`, `>`	`$ a=8; b=9`
		`$ if [[ ${a} < ${b} ]]; then echo $? ; fi`
		`0`

Typical File Tests

- (not) exists:
```
if [ ! -e "file" ] ; then echo "file does not exist" ; fi
```
- file is not zero:
```
[ -s "file" ] && (echo "file greater than zero")
```
- file is directory:
```
[ -d "file" ] && (echo "This is a directory")
```

- readable:
- writeable:
- executable:
```
[ -r "file" ]

[ -w "file" ]

[ -x "file" ]
```

- newer that file2:
```
[ "file" -nt "file2" ]
```

- Pitfalls when using variables:

wrong:
```
$ unset file_var; if [ -e ${file_var} ] ; then echo "yes" ; fi
yes
```

right:
```
$ unset file_var; if [ -e "${file_var}" ] ; then echo "yes" ; fi
```

for Loops

```
for arg in list
   do
    command
done
```

- Iterates command(s) until all arguments of *list* are passed
- *list* may contain globbing wildcards

- Example

```
#!/bin/bash

## Purpose: Loop over generated integer sequence
counter=1
for i in {1..10} ; do
   echo "loop no. ${counter}: ${i}"
   let counter+=1
done
```

while Loops

```
while condition
   do
    command
done
```

- Iterates command(s) as long as *condition* is true (or exit status 0)
- Allows indefinite loops

- Example

```
#!/bin/bash

## Purpose: Loop until max is reached
max=10
i=1
while (( ${max} >= ${i} )) ; do
   echo "${i}"
   let i+=1
done
```

Positional parameters

= Arguments passed to the script from the command line

Special variable	Meaning, notes
`$0`	Name of script itself
`$1, $2, $3`	First, second, and third argument
`${10}`	10th argument, but: `$10 = $1 + 0`
`$#`	Number of arguments
`$*`	List of all arguments as one single string
`$@`	List of all arguments, each argument separately quoted

- **Example:**

  Show differences between

  `${*}` and `${@}`

```
echo "Number PPs: ${#}"

i=1
for PP in "${@}" ; do
   printf "%3.3s.PP: %s\n" "${i}" "${PP}"
   let i+=1
done

i=1
for PP in "${*}" ; do
   printf "%3.3s.PP: %s\n" "${i}" "${PP}"
   let i+=1
done
```

`${WORKSHOP}/exercises/bash/special_var_01.sh`

Conditional evaluation - case

```
case variable in
    condition1)
        do_if_cond1_true/0
        ;;
    *)
        do_the_default
        ;;
esac
```

- analog to switch in C/C++
- to simplify multiple if/then/else
- each condition block ends with double semicolon
- If a condition tests true:

    a) commands in that block are executed

    b) case block terminates

Processing Input without getopts

- Combining: Positional parameter + case + while

```
${WORKSHOP}/exercises/bash/proc_input.sh
```

```
#!/bin/bash

## Purpose: Processing positional parameters

while (( ${#} > 0 )) ; do
   case "${1}" in

      ## script option: -h
      -h) echo "${1}: This option is for HELP"   ;;

      ## script option: -i + argument
      -i) echo "${1}: This option contains the argument ${2}"
          shift ;;


      ## default
       *) echo "${1}: This is non-defined PP"    ;;
   esac
   ## Shifting positional parameter one to the left: $1 <-- $2 <-- $3
   shift
done
```

Lifetime of Variables (1)

- Script execution:
  - assigned variables only known during runtime
  - assigned variables not known in „slave“ scripts until „exported“
  - Example:

```
${WORKSHOP}/exercises/bash/master_parse_var.sh
${WORKSHOP}/exercises/bash/slave_get_var.sh
```

```
#!/bin/bash

## Purpose: Demonstrate parsing of assigned variables

var1="Non-exported value of var1"
export var2="Exported value of var2"
slave_sh="./slave_get_var.sh"

## check if $slave_sh is executable for user
echo "${0}: \$var1 = $var1"
echo "${0}: \$var2 = $var2"
if [ -x "${slave_sh}" ] ; then
   "${slave_sh}"
fi
```

- But: export of variables in script to interactive shell session only via:

```
$ source script.sh
```

(compare ~/.bashrc)

Lifetime of Variables (2)

- Environmental variables

  a) can be read in e.g. `my_workDIR=${PWD}`

  b) during script changed, example:

```
...
## Purpose: Demonstrating effects on environmental variables

## Changing it during runtime
export HOME="new_home_dir"
echo "${0}: \${HOME} = ${HOME}"
...
```

`${WORKSHOP}/exercises/bash/env_var.sh`

```
$ echo ${HOME}; ./env_var.sh; echo ${HOME}
/home/kit/scc/ab1234
./env_var.sh: ${HOME} = new_home_dir
/home/kit/scc/ab1234
```

awk & sed: Command substitution

- **awk**

  → full-featured text processing language with a syntax reminiscent of C

  → use for complicated arithmetics or text or RE processing

  - Examples:

    a) logarithm of variable:

    ```
    a=10; echo ${a} | awk '{print log($1)}'
    ```

    b) first column reformated:

    ```
    awk '{printf "%20.20s\n",$1}' file
    ```

  - One-liners: http://awk.info/?OneLiners

- **sed**

  → non-interactive stream editor

  → use for deleting blank or commented lines etc

  - Example: delete all blank lines of a file:

    ```
    sed '/^$/d' file
    ```

  - One-liners: http://sed.sourceforge.net/sed1line.txt

Functions (1)

```
function my_name ()
{
    commands
}
```

- Stores a series of commands for ***later*** or ***repeative*** execution
- Functions are called by writing the name
- **Like scripts:** functions handle positional parameters
- Example:

`${WORKSHOP}/exercises/bash/fct_01.sh`

```
#!/bin/bash

## Purpose: Demonstrating features of functions

## Add to printf command a common string
function my_printf ()
{
   printf "${0}: $(date): ${@}"
}

my_printf "Hello World\n"
```

Functions (2)

- **`local`** variables: values do not exist outside function, example:

```
#!/bin/bash
## Purpose: Demonstrating features of functions

var1="global value"

## Function: assign to global var1 temporarily a local value
function locally_mod_var ()
{
   local var1=${1}
   if [ -z "${var1}" ] ; then
      return 1
   fi
   echo "fct: local \${var1} = ${var1}"
   var1="new value in fct"
   echo "fct: local \${var1} = ${var1}"
}

echo "main: global \${var1} = ${var1}"
locally_mod_var "${var1}"
echo "main: global \${var1} = ${var1}"
```

`${WORKSHOP}/exercises/bash/fct_02.sh`

- **`return`**: Terminates a function, optionally takes integer = „exit status of the function“

Thank you for your attention!

Regular Expression matching