

# Advanced Bash Scripting

Robert Barthel, SCC, KIT



UNIVERSITÄT  
HEIDELBERG  
ZUKUNFT  
SEIT 1386

**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**



# How to read the following slides

Abbreviation	Full meaning
<code>\$ command -opt value</code>	<code>\$</code> = <b>prompt</b> of the interactive shell The full prompt may look like: <code>user@machine:path \$</code> The <code>command</code> has been entered in the interactive shell session
<code>&lt;integer&gt;</code> <code>&lt;string&gt;</code>	<code>&lt;&gt;</code> = Placeholder for integer, string etc
<code>foo, bar</code>	Metasyntactic variables
<code>\${WORKSHOP}</code>	<code>/pfs/data1/software_uc1/bwhpc/kit/workshop/2019-10-09</code>

## Sources of this slides?

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

# Where to get the slides/exercises/reservation?

- [https://indico.scc.kit.edu/e/bwhpc\\_course\\_2019-10-09](https://indico.scc.kit.edu/e/bwhpc_course_2019-10-09) or  
bwUniCluster: /pfs/data1/software\_uc1/bwhpc/kit/workshop/2019-10-09

- Slides
- Exercises

- Workshop reservation:  
single node:

```
msub -A workshop  
-l advres=bwhpc-workshop_single.140
```

- multi node:

```
msub -A workshop  
-l advres=bwhpc-workshop_single.140
```

Überblick / Overview

Agenda

Registrierung / Registration

Formular / Form

Das Steinbuch Centre for High Performance Computing (HPC) is... (zukünftigen) Nutzen Landesforschungsh... Zugang und Nutzung vormittags an Einst... Teilnehmerzahl (35

The Steinbuch Centre for computing (HPC) is... (bwUniCluster, bw... about access and us... morning beginners... limited to 35. No co

Starts 6 Dec  
Ends 6 Dec  
Europe/Ber

Slides exercises

# How to do exercises?

- Login to cluster & Generate workspace „bwhpc-course“

```
$ ws_allocate bwhpc-course 30
Workspace created. Duration is 720 hours.
Further extensions available: 3
/pfs/work2/workspace/scratch/xy1234-bwhpc-course-0
```

- Copy examples to your workspace

```
$ WORKSHOP=/pfs/data1/software_uc1/bwhpc/kit/workshop/2019-10-09
$ cd $(ws_find bwhpc-course)
$ mkdir -v 2019-10-09; cd 2019-10-09
$ cp -pr ${WORKSHOP}/exercises/01 ./
```

- Submit jobs from your workspace

```
$ cd $(ws_find bwhpc-course)/2019-10-09/01
$ msub <jobscript>
```

# 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

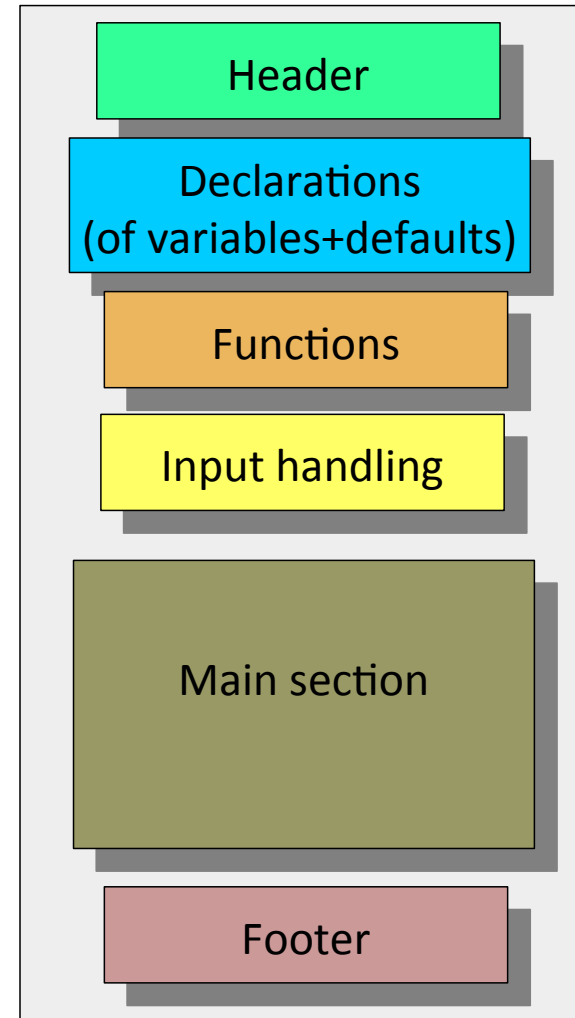
## ■ Bash almost everywhere installed

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



# Header & Line format

- Sha-Bang = '#!' (= activates interpreter directive)

→ 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:

```
;
```

Semicolon

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

- Group commands

- In current shell:

```
{ cmd1; cmd2; }
```

- In subshell:

```
(cmd1; cmd2)
```

# 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
- formatted output

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

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



# Special bash characters (1)

Chars with meta-meaning

## # 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}'
```

# Special bash characters (2)

Chars with meta-meaning

`$( )`

Command substitution

old version: ``` (backticks) → do not use anymore

```
$ echo "today = $(date)"  
today = Wed Oct 11 02:03:40 CEST 2017
```

`( )`

Group commands in a subshell

```
$ (ls -l; date)
```

`(( ))`

Double-parentheses construct

→ arithmetic expansion and evaluation

```
$ echo $((1 + 3))  
4
```

\$ prefixing of `(( ))` to return the value it holds

`[ ]`

Test builtin (cf. slide 24)

or

Array element (cf. slide 11)

# Globber

- = filename expansion  
→ recognices and expands „wildcards“
- but this is **not** a Regular Expression interpretation (for such use awk/sed)

## ■ 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!**  
→ check difference:
- Values can be generated by commands

```
var1=abcdef
```

```
var2=01_{var1}_gh
```

vs

```
var3=01_$var1_gh
```

```
var4=$(date)
```

```
$ echo {var2}
```

```
01_abcdef_gh
```

```
$ echo {var3}
```

```
01_
```

## ■ 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
```

```
$ declare -a arr=( '1 2' 3)  
$ echo {arr[0]}  
1 2
```

← space is separator

## Variables (2)

- declare – *cont.*

- Arrays: e.g. store file content in array:

- a) 1 element per string

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

=

```
a=( $(cat file) )
```

- b) 1 element for whole file

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

- c) 1 element per line

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

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

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

# Manipulation of Variables (1)

Syntax	Does?	Examples
<code>\${#var}</code>	String length	<code>\$ A='abcdef_abcd'; echo \${#A}</code> 11
<code>\${var:pos:len}</code>	Substring extraction:	
	a) via Parameterisation	<code>\$ POS=3; echo \${A:\${POS}:2}</code> de
	b) Indexing from right	<code>\$ echo \${A:(-2)}</code> cd
<code>\${var#sstr}</code>	Strip shortest match of \$sstr from front of \$var	<code>\$ sstr=a*b; echo \${A#\${sstr}}</code> cdef_abcd
<code>\${var%sstr}</code>	Strip shortest match of \$sstr from back of \$var	<code>\$ sstr=c*d; echo \${A%\${sstr}}</code> abcdef_ab
<code>\${var/sstr/repl}</code>	Replace first match of \$sstr with \$repl	<code>\$ sstr=ab; rp=AB; echo \${A/\${sstr}/\${rp}}</code> ABcdef_abcd
<code>\${var//sstr/repl}</code>	Replace all matches of \$sstr with \$repl	<code>\$ echo \${A//\${sstr}/\${rp}}</code> ABcdef_ABcd
<code>\${var/#sstr/repl}</code>	If \$sstr matches front-end, replace by \$repl	<code>\$ sstr=a; rp=z_; echo \${A/#\${sstr}/\${rp}}</code> z_bcdef_abcd
<code>\${var/%sstr/repl}</code>	If \$sstr matches back-end, replace by \$repl	<code>\$ sstr=d; rp=_z; echo \${A/%\${sstr}/\${rp}}</code> abcdef_abc_z

# Manipulation of Arrays

Syntax	Does?	Examples
<code>\${#array[@]}</code>	Number of elements	<pre>\$ dt=( \$(date) ); echo \${#dt[@]} 6</pre>
<code>\${array[@]:p1:p2}</code>	Print elements from no. <b>p1</b> to <b>p2</b> :	<pre>\$ echo \${dt[@]:1:2} Feb 25</pre>
<code>\${array[@]#sstr}</code>	Strip shortest match of \$sstr from front of all elements of Array	<pre>\$sstr=W*d; echo \${dt[@]#\${sstr}} Feb 25 10:18:22 CET 2015</pre>

## ■ 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
```

# Exercise 1: Variables and Arrays (5min)

- Write a bash script that:
    - Has a **variable** named **outputfile** which:
      - Is readonly and consists of 3 strings:
        1. Environment variable \$LOGNAME
        2. Arbitrary string of 4 characters generated in subshell via:  
`mktemp -u XXXX`
        3. First 2 characters of the current month (→ use „date“) using a bash array
- Hint:
- ```
array=$(date))
```



# Exercise 1: Variables and Arrays - Solution

- Write a bash script that:
  - Has a **variable** named **outputfile** which:
    - Is **readonly** and consists of 3 strings:
      1. Environment variable **\$LOGNAME**
      2. **Arbitrary string** generate in subshell via: (use „mktemp -u XXXX“)
      3. **First 2 characters** of the **current month** (→ use „date“) using a bash array

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

```
/${WORKSHOP}/solutions/01/exercise_1.sh
```

```
# in case language is en_us.utf8, month is 2.element in "date"
```

```
array=$(date)
```

```
month=${array[1]:0:2}
```

```
declare -r outputfile="${LOGNAME}_$(mktemp -u XXXX)_${month}.log"
```

```
echo ${outputfile}
```

```
# Try changing output file
```

```
outputfile="new"
```

# Output & Input Redirection (1)

| Syntax                                    | Does?                                                                                                             | Examples                                                                                                                   |
|-------------------------------------------|-------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------|
| <code>exe &gt; log</code>                 | Standard output ( <b>stdout</b> ) of application <code>exe</code> is (over)written to file <code>log</code>       | <code>\$ date &gt; log; cat log</code>                                                                                     |
| <code>exe &gt;&gt; log</code>             | Standard output ( <b>stdout</b> ) of application <code>exe</code> is append to file <code>log</code>              | <code>\$ date &gt;&gt; log; cat log</code>                                                                                 |
| <code>exe 2&gt; err</code>                | Standard output ( <b>stderr</b> ) of application <code>exe</code> is (over)written to file <code>err</code>       | <code>\$ date 2&gt; err; cat err</code>                                                                                    |
| <code>exe 2&gt;&gt; log</code>            | Standard output ( <b>stderr</b> ) of application <code>exe</code> is append to file <code>log</code>              | <code>\$ date 2&gt;&gt; err; cat err</code>                                                                                |
| <code>exe &gt;&gt; log 2&gt;&amp;1</code> | Redirects <b>stderr</b> to <b>stdout</b>                                                                          | <code>\$ date &gt;&gt; log 2&gt;&amp;1</code>                                                                              |
| <code>exe1   exe2</code>                  | Passes <b>stdout</b> of <code>exe1</code> to standard input ( <b>stdin</b> ) of <code>exe2</code> of next command | <i># Print stdout &amp; stderr to screen and then append both to file</i><br><code>\$ date 2&gt;&amp;1   tee -a log</code> |
| <code>exe &lt; inp</code>                 | Accept <b>stdin</b> from file <code>inp</code>                                                                    | <code>\$ wc -l &lt; file</code>                                                                                            |

## Output & Input Redirection (2)

### ■ Take care of order when using redirecting

#### ■ e.g:

```
(ls -yz; date) >> log 2>&1
```

- Stdout (date) redirected to file „log“
- Stderr (invalid option of ls) redirected to file pointed to by stdout

```
(ls -yz; date) 2>&1 >> log2
```

- Stderr (invalid option of ls) redirected to stdout (channel), but not written file
- Stdout (date) redirected to file

### ■ Suppressing stderr

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

Usage? Keep variable empty when error occurs

- e.g. list of files with extension log

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

- if no files with extension log exist, \$list\_logs is empty

## 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.

# Output & Input Redirection (4)

## ■ Reading input e.g. file line by line

```
#!/bin/bash ${WORKSHOP}/exercises/01/01_read_input.sh  
  
declare -i i=1  
while read -r line ; do  
    echo "line ${i}: ${line}"  
    let i+=1  
done < 01_input_file
```

## ■ Reading output of other commands line by line, e.g. „ls -l”

```
#!/bin/bash ${WORKSHOP}/exercises/01/02_read_input.sh  
  
declare -i i=1  
while read -r line ; do  
    echo "line ${i}: ${line}"  
    let i+=1  
done < <(ls -l *)
```

### Process substitution:

a form of redirection;  
input/output of process = temp file

## Manipulation of Variables (2)

### ■ Example:

```
#!/bin/bash ${WORKSHOP}/exercises/01/03_var_manipulation.sh  
  
## Purpose: Define automatic output names for executables  
  
exe="03_binary.x"  
  
## Assume: $exe contains extension .x or .exe etc  
sstr=".*" ## substitution string  
log="${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                                                                                                                         |
|--------------------------------------|------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------|
| <code>\${var-<b>-\${def}</b>}</code> | If \$var not set, set value of \$def                                                     | <pre>\$ unset var; def=new; echo \${var-<b>-\${def}</b>}</pre> <code>new</code>                                                  |
| <code>\${var:<b>-</b>-\${def}</code> | If \$var not set or <i>is empty</i> , set value of \$def                                 | <pre>\$ var=''; def=new; echo \${var:<b>-</b>-\${def}}</pre> <code>new</code>                                                    |
|                                      |                                                                                          | <pre><b># Job Id for interactive and MOAB runs</b><br/>jobID=\${MOAB_JOBID:<b>-</b>-\${BASHPID}}</pre>                           |
| <code>\${var:<b>?</b>-\${err}</code> | If \$var not set or <i>is empty</i> , print \$err and abort script with exit status of 1 | <pre>\$ var=''; err='ERROR - var not set'<br/>\$ echo \${var:<b>?</b>-\${err}}</pre> <code>bash: var: ERROR - var not set</code> |

# 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 2>/dev/null; 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                                                                              | <pre>\$ if (( 2 &gt; 0 )) ; then echo yes ; fi yes</pre>                                                              |
| [ ]       | Part of (file) test builtin,<br>arithmetic evaluation only<br>with -gt, -ge, -eq,<br>-lt, -le, -ne | <pre>\$ if [ 2 -gt 0 ] ; then echo yes ; fi yes \$ # existence of file \$ if [ -e "file" ] ; then echo yes ; fi</pre> |
| [[ ]]     | Extended test builtin;<br>allows usage of<br>&&,   , <, >                                          | <pre>\$ a=8; b=9 \$ if [[ \${a} &lt; \${b} ]]; then echo \$? ; fi 0</pre>                                             |

# 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:

```
[ -r "file" ]
```

■ writeable:

```
[ -w "file" ]
```

■ executable:

```
[ -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 1

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

## ■ Example 2

```
## Example 2: Loop over space separated list of strings
list="file_1,file_2,file_3"
for x in ${list//,/ " "} ; do
    echo ${x}
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 (1)

= 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, <b>but if not brace protected</b> : \$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/01/04_special_var.sh`

## Positional parameters (2)

= 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 if not brace protected: \$10 = \$1 + 0 |
| \$#              | Number of arguments                                       |
| \$*              | List of all arguments as one single string                |
| @                | List of all arguments, each argument separately quoted    |

Shifting positions:

**shift**

Drops \$1 → shifts \$2 to \$1 → \$3 to \$2 and so → \$# is reduced by 1

# 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

```
#!/bin/bash ${WORKSHOP}/exercises/01/05_case.sh

## Purpose: Color output red or blue
e0="\033[0m"; eR="\033[31;1m"; eB="\033[34;1m"
case ${1} in
    red)
        echo -e "${eR}This line is red${e0}"
        ;;
    blue)
        echo -e "${eB}This line is blue${e0}"
        ;;
    *)
        echo -e "Line wo color" ;;
esac
```

## Exercise 2 (10-15 min)

- Write Script that processes options:

-h

-i <integer>

without shell build getopt combining „positional parameter“, „shift“, „tests“, „case“ and „while“

Template: `${WORKSHOP}/exercises/01/06_proc_input.sh`

→ Replace everything betw. ... and ... by code

```
#!/bin/bash

while ...test total num_positional parameter (PP) greater zero... ; do
  case "PP1" in
    ## script option: -h
    ...PP is option1...) ...echo something...
    ;;
    ...PP is option2...) ...echo PP2...
    ...do PP shift...
    ;;
  esac
  ...do PP shift...
done
```



# Exercise 1: Solution

## Processing Input without *getopts*

- Combining: Positional parameter + shift + tests + case + while

```
#!/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
```

```
/${WORKSHOP}/solutions/01/06_proc_input.sh
```

# Lifetime of Variables (1)

## ■ Script execution:

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

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

```
${WORKSHOP}/exercises/01/07_master_parse_var.sh  
${WORKSHOP}/exercises/01/08_slave_get_var.sh
```

```
## Purpose: Demonstrate parsing of assigned variables
```

```
var1="Non-exported value of var1"
```

```
export var2="Exported value of var2"
```

```
slave_sh="./08_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/01/09_env_var.sh
```

```
$ echo ${HOME}; ./06_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 *regular expression* processing

### ■ Examples:

a) logarithm of variable:

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

b) **print first col. reformated:**

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

■ One-liners: <http://www.pement.org/awk/awk1line.txt>

## ■ 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 **repetitive** execution
- Functions are called by writing the **name**
- Functions also process positional parameters
- Example:

```
#!/bin/bash  
  
## Purpose: Demonstrating features of functions  
## Add to printf command the date string  
function my_printf ()  
{  
    printf "${0}: $(date): ${@}"  
}  
  
my_printf "Hello World\n"
```

```
./10_fct.sh: Tue Apr 9 19:57:40 CEST 2019: Hello World
```

# 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/01/11_fct.sh`

- **return**: Terminates a function, optionally takes integer = „exit status of the function“, do not use „exit“ integers above 255 (cf. slide 23).

# Trap

- Catch abort signals, e.g.
  - SIGHUP = Hangup
  - SIGINT = Interrupt (Ctrl + C)
  - SIGTERM = Termination signal (kill -15)

and do something (e.g. cleanup) before abort

- Example:

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

```
${WORKSHOP}/exercises/01/12_trap.sh
```

```
cleanup(){  
    echo "Cleanup before interrupt and exit"  
    exit 0  
}
```

```
## Trap interrupt with function cleanup  
trap "cleanup" SIGINT
```

```
## Loop forever doing not really anything  
while true ; do  
    sleep 10  
done
```

Thank you for your attention!