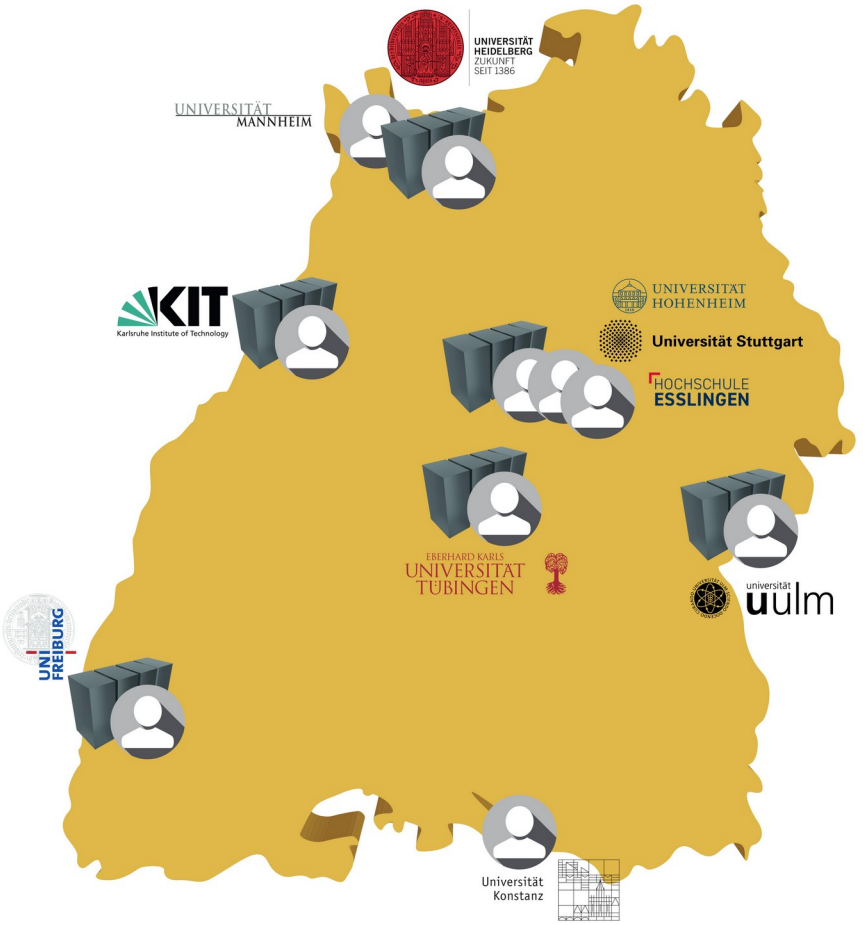
bw|HPC

KIT | NHR
National High-Performance Computing Center

Advanced Bash Scripting

Michele Mesiti, SCC, KIT *



** Original materials developed by others at SCC, KIT*

Funding: Baden-Württemberg MINISTERIUM FÜR WISSENSCHAFT, FORSCHUNG UND KUNST · Bundesministerium für Bildung und Forschung

How to read the following slides

Abbreviation	Full meaning
`$ command -opt 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}`	`@uc2:/opt/bwhpc/common/workshops/2024-10-17` `@hk:/software/all/workshops/2024-10-17`

Sources of this slides?

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

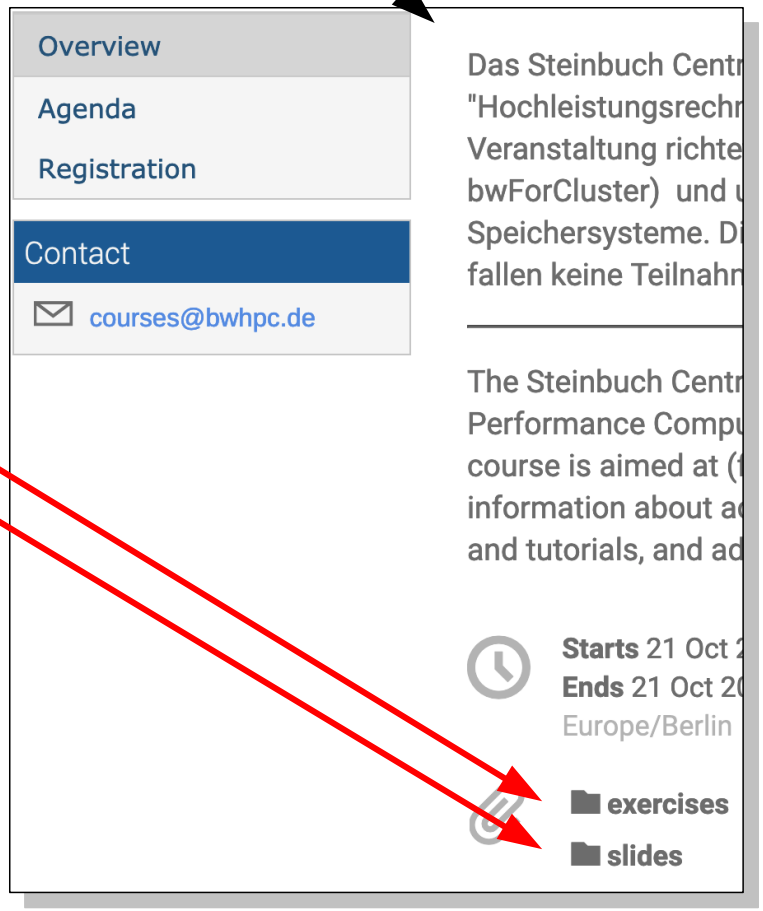
Where to get the slides/exercises/reservation?

- https://indico.scc.kit.edu/e/hpc_course_2024-10-17 or

  bwUniCluster: /opt/bwhpc/common/workshops/2024-10-17

  horeka:/software/all/workshops/2024-10-17

  - exercises
  - slides



- Workshop reservation – mult inode:
  - bwUniCluster 2.0

```
sbatch --reservation=ws
```

  - HoreKa

```
sbatch --reservation=ws
```

How to do excercises?

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

```
$ ws_allocate bwhpc-course 30
Info: creating workspace
/pfs/work7/workspace/scratch/ab1234-bwhpc-course
remaining extensions  : 3
remaining time in days: 30
```

- Copy examples to your workspace

```
$ WORKSHOP=/opt/bwhpc/common/workshops/2024-10-17
$ cd $(ws_find bwhpc-course)
$ mkdir -v 2024-10-17; cd 2024-10-17
$ cp -vpr ${WORKSHOP}/exercises/01  ./
```

- Submit jobs from your workspace

```
$ cd $(ws_find bwhpc-course)/2024-10-17/01
$ sbatch -p {single|cpuonly} --reservation=ws <jobscript>
```

Why (not) Bash?!

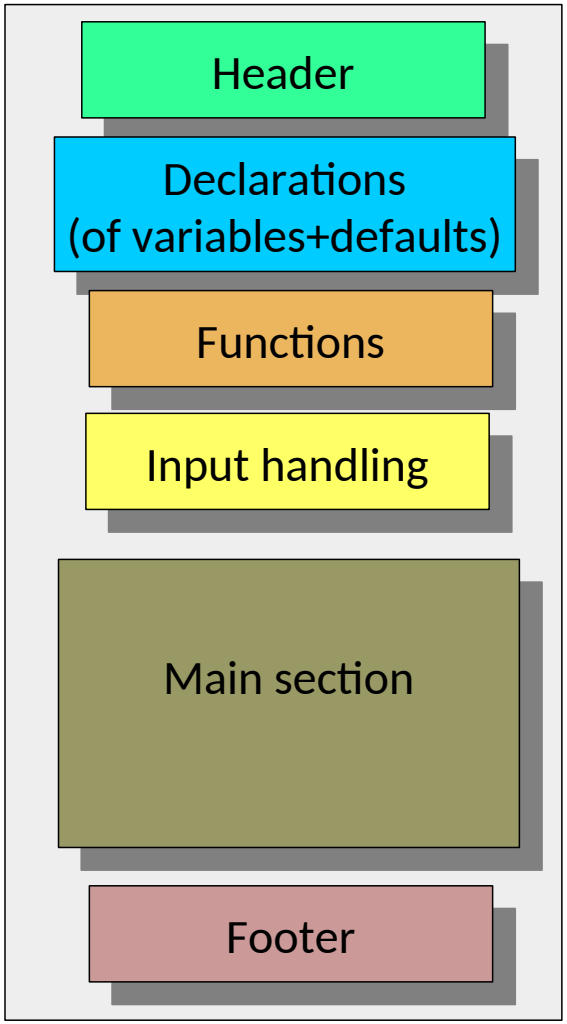
- **Great at:**
  - **managing batch jobs**
  - managing external programs
  - invoking entire UNIX command stack & many builtins
  - Powerful scripting language
  - Portable and version-stable
  - almost everywhere installed

- **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

- Hash-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:*

```
$ echo $BASHPID; { echo $BASHPID; sleep 10; }
```

  - *In subshell:*

```
$ echo $BASHPID; ( echo $BASHPID; sleep 10 )
```

Bash Output

- **echo**

  a) by default, trails every output with a „newline“

  b) prevent newline with -n:

  c) parsing „escape sequences“ with -e:

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

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

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

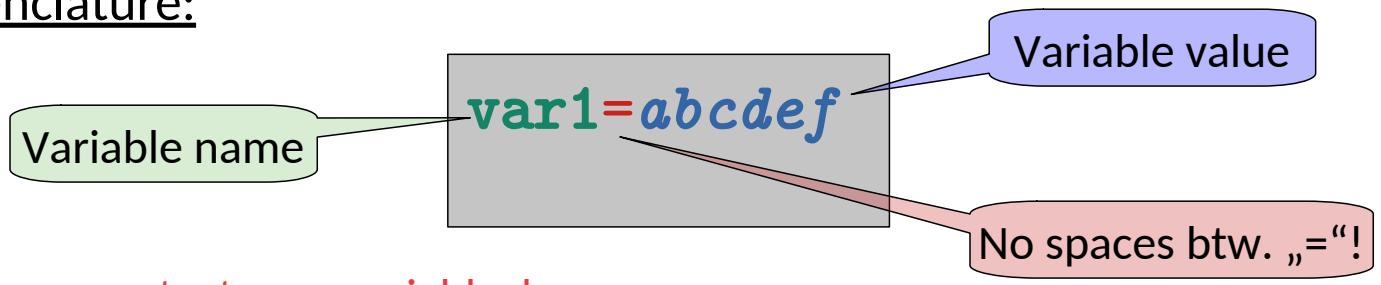
- **printf** = „enhanced“ echo
  - by default no „newline“ trailing
  - formated output

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

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

Variables (1)

- Nomenclature:



  - Brace protect your variables!

    → check difference:

```
var2=01_${var1}_gh
        vs
var3=01_$var1_gh
```

```
$ echo ${var2}
01_abcdef_gh
$ echo ${var3}
01_
```

  - Values can be generated by commands:

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

- 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

Variables (2)

- Declare
  - 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

  - *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 $ when:

assign
declare
export
unset

```
a="value"
declare -i a=41
export a
unset a
```

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 subsitution

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

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

`(  )` *Group commands in a subshell*
(or creates an array)

```
$ (ls -l; date)
$ arr=(1 2 3)
```

`((  ))` Double-parentheses construct

→ arithmetic expansion and evaluation

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

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

`[  ]` Test builtin (cf. slide 25)
or
Array element (cf. slide 12)

Globbing (pathname expansion)

- → recognizes and expands „wildcards“
- but this is **not** a Regular Expression interpreter (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

  `^` = to negate the wildcard match

  e.g. list all files not starting with a

```
$ ls [a,b]*
```

```
$ ls [^a]*
```

Manipulation of Variables (parameter expansion)

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 front-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 (including parameter expansion)

Syntax	Does?	Examples
`${#array[@]}`	Number of elements	`$ dt=( $(date) ); echo ${#dt[@]}` `6`
`${array[@]:start:n}`	Print **n** elements starting from **start**:	`$ 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
```

Exercise 1: Variables and Arrays

- 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

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

```
#!/bin/bash

# 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
`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)

- ***Bonus:*** 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

Bonus: 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

${WORKSHOP}/exercises/01/01_read_input.sh

```
#!/bin/bash

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`"

${WORKSHOP}/exercises/01/02_read_input.sh

```
#!/bin/bash

declare -i i=1
while read -r line ; do
   echo "line ${i}: ${line}"
   let i+=1
done < <(ls -l *)
```

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

Manipulation of Variables (2)

- Example:

${WORKSHOP}/exercises/01/03_var_manipulation.sh

```
#!/bin/bash

## 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}
```

Bonus: 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`
		`# Job Id for interactive and Slurm runs` `jobID=${SLURM_JOB_ID:-${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

```
#!/bin/bash
echo "printed line"
exit
echo "not printed line"
```

- Every command returns an exit status
  - successful = 0
  - non-successful > 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	`$ 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`
		`$ # existence 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 (see man [ )

- **(not) exists**:

```
if [ ! -e "file" ]; then echo "file does not exist" ; fi
```

- file is not empty:

```
if [ -s "file" ]; then echo "file size > zero" ; fi
```

- file is directory:

```
if [ -d "file" ]; then echo "This is a directory" ; fi
```

- readable:

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

- writeable:

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

- executable:

```
[ -x "file" ]
```

- newer than 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
```

Bonus: 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, 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

Example:

Show differences between

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

`${WORKSHOP}/exercises/01/04_special_var.sh`

```
echo "Number
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
```

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

Bonus: conditional evaluation and pattern matching - case

```
case var in
   pattern1)
      do_if_var_matches_pattern1
      ;;
   *)
      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

${WORKSHOP}/exercises/01/05_case.sh

```
#!/bin/bash

## 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
```

Bonus: Excercise 2

- Write Script that processes options:

  `-h`

  `-i <integer>`

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

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

→ Replace everything between ... 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
```

Bonus: Solution of Excercise 2
Processing Input without *getopts*

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

${WORKSHOP}/solutions/01/06_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
```

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: https://github.com/jweslley/dotfiles/blob/master/docs/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:

${WORKSHOP}/exercises/01/10_fct.sh

```
#!/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: Mon Oct 25 11:57:40 CEST 2021: 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 24).

Trap

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

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

- Example:

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

```
#!/bin/bash

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

## Trap interrupt with funtion cleanup
trap "cleanup" SIGINT

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

Lifetime of Variables (1)

- During **script** execution:
  - assigned variables only known during runtime
  - assigned variables not known in „worker“ scripts until „exported“
  - Example:

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

```
#!/bin/bash

## Purpose: Demonstrate parsing of assigned variables

var1="Non-exported value of var1"
export var2="Exported value of var2"
worker_sh="./08_worker_get_var.sh"

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

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

```
$ source script.sh
```

ompare ~/.bashrc)

Bonus: 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
```

Some Additional Advice

- Use a linter: download shellcheck or use it from the web
- *Consider* making your script fail on error like a program would do, e.g.:
  - exit immediately if a command exits with a non-zero status:

```
$ set -e
```

  - Treat unset variables as an error when substituting

```
$ set -u
```

  - Make pipelines fail if commands in them fail:

```
$ set -o pipefail
```

- You can even unit test your shell scripts with BATS

Thank you for your attention!