Bash

Week 27-29/10

MA030366 (Term 2) R2-B5-2026

Zacharov Igor

i.zacharov@skoltech.ru

Bash topics

- Variables
- Arrays
- Using Linux commands in assignments
  - Assignment to variables
  - Arithmetic operators
- Reading input and writing output
- Functions
  - Positional arguments
- Comparisons and Conditions
- Control flow
  - Loops
- Running bash scripts

Variables

- Variables store data, alter & reuse them throughout the script
  - Ex.: `var1=“string”`
    - Note: **no spaces between equal sign and the operands!**
    - Note: **use mostly lower case characters, internal shell variables often use capitals**
- Reuse of data in variables with **$**
  - Ex.: echo $var1
  - The strings with “ (double quotes) fully expands the variables and wild chars
    - Eg. echo “this is a variable data: $var1”
  - The strings with ‘ (single quotes) expands variables only (no wild chars)
- Creation of new variable names based on previous names
  - Use **{}** to isolate the previously defined name
  - Ex.: var2=new_${var1}_name
    - Note: ${parameter} expands the variable name within the ${…} part before assignment

Bash arrays

- Arrays are set of elements under a single name, define with **()**
  - Ex. Definition:

  allThreads=(1 2 4 8 “string” 12 20)
  - Extend arrays:

   allThreads+=(“a” 16 64 “b”)
  - Access elements with [] as follows: var=${allThreads[3]}
    - Note: array indexes start from zero (0).
  - Print full array:

  echo ${allThreads[@]}
    - Note: the sign @ stands for all
  - To get the array indices:

  echo ${!allThreads[@]}

`arr=()`	Create empty array
`arr=(1 2 3)`	Initialize array
`${arr[2]}`	Retrieve 3rd element
`${arr[@]}`	Retrieve all elements
`${!arr[@]}`	Retrieve array indices
`${#arr[@]}`	Calculate array size
`arr[0]=3`	Set 1st element
`arr+=(4 5)`	Append value(s)
`str=$(ls)`	Save ls output in string
`arr=( $(ls) )`	Save ls output as array
`${arr[@]:s:n}`	Retrieve n elements starting at index s

Command assignment to variables

- Output of any Linux command can be assigned to a variable with **$()** construct or with `` (back quotes)
  – Ex. user=$(whoami)
  – Ex. user=`whoami`
    - Note, the $() construct produces separate shell to run the command
- Arithmetic operations: use **(())** double brackets for integer expressions
  – Ex. val1=$((10*5+15)); echo $val1
  – Ex. val2=$((7+3-$val1)); echo $val2
  – Ex. val3=$(($val2+100)); ((val3++)); echo $val3
- Arithmetic operations: use **bc** command for floating point expressions
  – Ex. echo “55/3” | bc
  – Ex. echo “scale=2; 55/3” | bc

Input and Output redirection

- The input and output can be redirected with **<** and **>** respectively
  - Ex.: `cat myfile > yourfile` will re-direct the output of the cat command
  - Ex.: cat <myfile will redirect standard input to read myfile
  - Note: when reading stdin the end of input is given by ^D (CTRL+D keys)
- The **read** command will take terminal input
  - Ex.: read xx will wait for terminal line (until ENTER) and assign to xx
  - Ex.: read with no var name will assign to $REPLY
  - Note: read is bash buildin, see man read (eg. -p –s flags for prompt handling)

```
#!/bin/bash

cat <<EOF >out
Line
Line
EOF
```

- The redirection of stderr with: **2>**
- The redirection of both, stderr and stdout with: **&>**
  - Note: /dev/null acts as a sink, ex.: `cat myfile 2>/dev/null`
- Herein document with **<<EOF** and finishes with **EOF** on a line
  - Note EOF can be any other word as an end marker

Functions

- Functions allow to reuse code
  - grouping number of different commands into a single command
  - Defined by **`function`** ***`name`*** **`() { … }`**
    - Note: the () in the definition are optional
  - The functions can be called after they are defined
  - The function refers to positional arguments, i.e. **$1 $2** etc.
    - Note **`$0`** is the name of the function
    - Note **$#** number of arguments
    - Note **$*** means all arguments
  - Modifications of positional arguments:
    - ${1:-default} if not set, use default
    - ${1:=default} if not set, set to default
    - ${1:+altval} if set, use altval, else use null string
    - ${1:?err_msg} if set use it, else print err_msg and exit(1)

```
#!/bin/bash
in="string"

function new_function {
        echo $1
}

new_function $in
```

https://tldp.org/LDP/abs/html/parameter-substitution.html

Comparisons and Conditions

- The comparison is performed with **[]** it gives status 0 is true, 1 is false
  - Ex.: a=2; b=3; [$a –lt $b]; echo $?
    - Will give 0 as output (true)
  - Ex.: [“apples” = “oranges”] ; echo $?
    - Will give 1 as output (false)

Description	Numeric	string
Less than	-lt	<
Greater than	-gt	>
Equal	-eq	=
Not equal	-ne	!=
Less or equal	-le	
Greater or equal	-ge	

- Conditional expression **if then else fi**
  - Ex.

```
if [ $a –lt $b ] ; then
          echo “$a is less than $b”
else
          echo “$b is less than $a”
fi
```

- Conditional expression **case esac**
  - Ex.

```
case $word in
          choice1|choice2|choice3)  echo yes;;
          choice4|choice5)          echo  no;;
          *)                        echo  default;;
esac
```

- Logical combinations:
  - Or: at least one must be true to evaluate true: `expression1 || expression2`
  - And: both must be true to evaluate to true: `expression1 && expression2`
  - Negation: if expr is false it evaluates to true: `! expression`

Loops

- The syntax for the various constructs
  - Until: execute body as long as test-cmds has status not zero

```
Until test-cmds; do body; done
```

  - While: execute body as long as test-cmds has status zero

```
while test-cmds; do body; done
```

  - For: expand words and execute body for each member of words

```
for name in words…; do body; done
```

  - For: evaluate arithmetic expressions and execute body as long as expr2 is evaluated to zero

```
for (( expr1; expr2; expr3 )); do body; done
```

- Notes:
  - Note: the ; (semicolon) can be replaced by new line
  - Note: break and continue builtins can be used to control the loops
  - Note: the status of the loop is status of the last command in the body

https://www.gnu.org/software/bash/manual/html_node/Looping-Constructs.html

Running bash scripts

- Shell script is a list of commands that can be executed automatically, when the script is started
  - The bash script is starting with #!/bin/bash
    - The path is the result of the command: `which bash`
  - Lines starting with # and text after # is treated like comments
  - The file containing the commands should have .sh extension
  - The file should have execution permission
    - For starting the script by stating its name, i.e `./script.sh`

```
#!/bin/bash

echo "Hello, World"
```

```
-rwxrw-r-- 1 i.zacharov i.zacharov 25 Oct 27 19:12 mycom.sh
```

  - It is possible to run the script with the command: `bash script.sh`
  - Debugging the script with –x and/or –v flags (i.e. `bash –x script.sh`)
    - The flag –x produces a trace of the statements in the script
    - The flag –v is verbose display of commands and their output
- Buildin are commands that do not start a separate shell for exec

– Note specials:	:	;	..	&
– Note wildcards:	*	[]	^	\