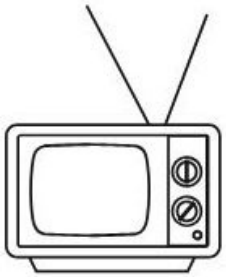

Previously On... CIS4930

- M1 – Basics
  - Aliases, filesystem, processes & signals
- M2 – Serious CLI
  - Shell quoting & escaping, Bash initialization files
  - Globbing, variables expansion, command expansion…
- M3 – Linux way
  - Redirections, piping, filters
- M4 – Regular Expressions
  - More filters: grep / egrep, Both Basic & Extended RegExs

M5 – Bash Scripting

- Special variables
- Arithmetic
- Conditionals
- Iteratives

Topic

Transforming Bash scripts' variables

Unfinished business

https://tldp.org/LDP/Bash-Beginners-Guide/html/sect_10_03.html

Substituting Variables

Bash variable	Meaning
`${VAR:-WORD}`	If VAR is not defined or null, the expansion of WORD is substituted; otherwise, the value of VAR is substituted

```
echo ${TEST:-test}
test
echo $TEST

export TEST=a_string
echo ${TEST:-test}
a_string
echo ${TEST2:-$TEST}
a_string
```

Substituting Variables

Bash variable	Meaning
`${VAR:=WORD}`	If the hyphen (-) is replaced with the equal sign (=), the value is assigned to the parameter if it does not exist

```
echo $TEST2


echo ${TEST2:=$TEST}
a_string

echo $TEST2
a_string
```

Substituting Variables

Bash variable	Meaning
`${VAR:?WORD}`	If it is not set, the expansion of WORD is printed to standard out and non-interactive shells quit.

```
#!/bin/bash
# This script tests whether a
variable is set. If not, it exits
printing a message.

echo ${TESTVAR:?"Missing Variable"}
echo "TESTVAR is set, proceed."
```

```
./vartest.sh
./vartest.sh: line 6: TESTVAR:
Missing Variable

export TESTVAR=present

./vartest.sh
present
TESTVAR is set, proceed.
```

Removing Substrings

Bash variable	Meaning
`${VAR:OFFSET:LENGTH}`	Strips a number of characters, equal to OFFSET, from a variable. The LENGTH parameter defines how many characters to keep, starting from the first character after the offset point. If LENGTH is omitted, the remainder of the variable content is taken

```
export STRING="thisisaverylongname"

echo ${STRING:4}
isaverylongname

echo ${STRING:6:5}
avery
```

Removing Substrings

Bash variable	Meaning

`${VAR#WORD}`
 	
`${VAR##WORD}`
	Deletes the pattern matching the expansion of WORD in VAR.
	
	WORD is expanded to produce a pattern just as in file name expansion.
	
	If the pattern matches the beginning of the expanded value of VAR, then the result of the expansion is the expanded value of VAR with the shortest matching pattern ("#") or the longest matching pattern ("##").

```
VAR="thisisaverylongname"

echo ${VAR}
thisisaverylongname

echo ${VAR#t*s}
isaverylongname

echo ${VAR##t*s}
averylongname
```

Removing Substrings

Bash variable	Meaning

`${VAR%WORD}`
 	
`${VAR%%WORD}`
	Deletes the pattern matching the expansion of WORD in VAR.
	
	WORD is expanded to produce a pattern just as in file name expansion.
	
	If the pattern matches the end of the expanded value of VAR, then the result of the expansion is the expanded value of VAR with the shortest matching pattern ("#") or the longest matching pattern ("##").

```
VAR="thisisaverylongname"

echo ${VAR}
thisisaverylongname

echo ${VAR%n*e}
thisisaverylong

echo ${VAR%%n*e}
thisisaverylo
```

Replacing Substrings

Bash variable	Meaning

`${VAR/PATTERN/STRING}`
 	
`${VAR//PATTERN/STRING}`
	Replaces only the first match (/) or all matches (//) of a pattern in VAR by STRING

```
VAR="thisisaverylongname"

echo ${VAR}
thisisaverylongname



echo ${VAR/name/string}
thisisaverylongstring



VAR="wootwootwootwoot"

echo ${VAR/oo/ha}
whatwootwootwoot

echo ${VAR//oo/ha}
whatwhatwhatwhat
```

Topic

Bash arrays

Unfinished business

Syntax

```
for i in ${!ARR[@]}
do
  echo $i " --> " ${ARR[$i]}
done
```

Syntax	Meaning
`arr=()`	Create an empty array
`arr=(1 2 3)`	Initialize array
`${arr[2]}`	Retrieve third element
`${arr[@]}`	Retrieve all elements
`${!arr[@]}`	Retrieve array indices
`${#arr[@]}`	Calculate array size
`arr[0]=3`	Overwrite 1st element
`arr+=(4)`	Append value(s)
`str=$(ls)`	Save ls output as a string
`arr=( $(ls) )`	Save ls output as an array of files
`${arr[@]:s:n}`	Retrieve n elements starting at index s

https://opensource.com/article/18/5/you-dont-know-bash-intro-bash-arrays

Length of an array

Bash variable	Meaning
`${#ARR[@]}`	Provides the length of the array.
`${#MyString}`	Works also on regular variables; provides the length of the string in characters

```
ARRAY=("one" "two" "one"
"three" "one" "four")

echo ${#ARRAY}
6



echo $SHELL
/bin/bash

echo ${#SHELL}
9
```

Removing Substrings in arrays

Bash variable	Meaning
`${*#WORD}` `${@#WORD}`	**Addendum** - With * or @, the pattern removal operation is applied to each positional parameter in turn, and the expansion is the resultant list.
`${DATA[*]#WORD}` `${DATA[@]#WORD}`	- If VAR is an array variable subscribed with "*" or "@", the pattern removal operation is applied to each member of the array in turn, and the expansion is the resultant list.
Same with `##` `%` `%%`	

```
declare -a ARRAY
ARRAY=("one" "two" "one"
"three" "one" "four")

echo ${ARRAY[*]#one}
two three four

echo ${ARRAY[*]#t}
one wo one hree one four

echo ${ARRAY[*]#t*}
one wo one hree one four

echo ${ARRAY[*]##t*}
one one one four
```

Topic

T1.1 – Conditional Statements

https://youtu.be/y9dxg6oviSo

Exit statuses

```
ls something_that_exists
echo $?
0
      # 0 means success (0 is OK)
      # success means true



ls something_that_does_not_exists
ls: cannot access 'something_that_does_not_exists': No such file or directory
echo $?
2
      # not 0 means failure
      # failure means false
      # we use a number to specify what happened
```

Our first `if` statement

- The `if` statement runs a command, then determines whether its exit status was a success or a failure

```
if ls $1
then
    echo "Everything went fine"
else
    echo "Oops, there was a problem"
fi
```

- The `else` component is optional

The elif statement

```
if  ls $1
then
    echo "Found 1st argument"
elif ls $2
then
    echo "Found 2nd argument"
else
    echo "Oops, there was a problem"
fi
```

&& and || used to chain commands (déjà vu)

```
ls something_that_exists && echo "it worked!"
# it worked!

ls something_that_exists || echo "it did not work"
# echo not triggered

ls something_that_does_not_exists && echo "it worked!"
# echo not triggered

ls something_that_does_not_exists || echo "it did not
work"
# it did not work
```

Our 3rd `if` statement

```
if ls $1 && touch $1
then
    echo "Everything went fine"
else
    echo "Oops, there was a problem"
fi
```

Topic

T1.2 – Test Command

https://youtu.be/D43xWB4h_yY

Comparing numbers

- `if` is meant to run a command
- So we need a command to evaluate Boolean expressions :)

Operator	Meaning
`-gt`	Integer comparisons: >
`-ge`	>=
`-lt`	<
`-le`	<=
`-eq`	==

```
test 25 -gt 20 ; echo $?
0
test 25 -ge 40 ; echo $?
1

test 0025 -eq 25 ; echo $?
0 #integers comparison
```

Comparing Strings

- Use the right comparisons for the right data types…

Operator	Meaning
=	String equality
!=	String differences
\<	Lexicographic order
\>	Lexicographic order
-z	Null string test
-n	String not empty test

```
test 0025 -eq 25 ; echo $?
0 #integers comparison
test 0025 = 25 ; echo $?
1 #strings comparison

test 40 -eq 40 ; echo $?
0 #integers comparison
test "40" -eq "40" ; echo $?
0

test 40 != 40 ; echo $?
1
test 0040 != 40 ; echo $?
0
```

Testing files

File operator	Meaning
`-e -a`	Test if file or folder exists
`-r`	File or folder exists & has read permissions
`-w`	Same for write permissions
`-x`	Same for execute permissions
`-d`	Test if target is a folder
`-h -L`	Test if target is a link
`-s`	Not empty
`-N`	Modified since last read
`-nt`	File1 is newer than file2
`-ot`	File1 is older than file2

```
test -e file_right_here
0

test -d folder_right_here
0
```

Using `test` w/ `if` statements: 2 syntaxes

```
#!/bin/bash

if   test -d $1
then
     echo "$1 is a folder"
elif test -e $1
then
     echo "$1 is a file"
else
     echo "$1 is weird :)"
fi
```

```
#!/bin/bash

if   [ -d $1 ]
then
     echo "$1 is a folder"
elif [ -e $1 ]
then
     echo "$1 is a file"
else
     echo "$1 is weird :)"
fi
```

Logical Operators & Grouping

Operator	Meaning
-a	AND
-o	OR

- Not recommended due to quirks in the implementation of these logical operators.
- Instead, use && and || at the shell level

```
# multiple test commands w/ shell logical operators
if [ "$varA" = 1 ] && { [ "$varB" = "t1" ] || [ "$varC" = "t2" ]; };
then
      do-something
fi

# -a and -o operators for test command
if [ "$varA" = 1 -a \( "$varB" = "t1" -o "$varB" = "t2" \) ]
then
      do-something
fi
```

https://stackoverflow.com/questions/6270440/simple-logical-operators-in-bash

Watchout for the syntax:

```
#!/bin/bash

if    [ -d $1 ] && [ -d $2 ]
then
      echo "$1 and $2 are folders"
elif  [ -e $1 ] || [ -r $2 ]
then
      echo "$1 exists, $2 can be read"
else
      echo "This is weird :)"
fi
```

```
[ -e myfile && -r myfile ]
# nope, [] only around tests

[ -e myfile ] && [ -r myfile ]
# yup
```

External Cmds AND builtins available

- Why do we have both?
- Probably for other shells (sh?)

```
type test
test is a builtin
type [
[ is a builtin
```

```
ls /bin/test /bin/[
/bin/test
/bin/[
```

https://stackoverflow.com/questions/53364895/why-are-some-bash-commands-both-built-in-and-external

Alternative way to evaluate conditions: (( ))

- This is an alternative way to evaluate conditions
- Shell arithmetic expansion set the exit status to
  - 1 if the expression evaluates to 0
  - 0 otherwise
- Notations are simpler (no need to escape operators)

Another Alternative [[ ]]

- Less portable across shells: non posix, originally ksh extension
- adopted as Bash keyword (vs. command for test and [ ])
- Keyword → more convenient syntax: < not redirection, ( ) not subshell
- == and = are equivalent but former is a bash extension

[[ ]]


```
# From previous slides:

if [ "$varA" = 1 ] && { [ "$varB" = "t1" ] || [ "$varC" = "t2" ]; };
…

if [ "$varA" = 1 -a \( "$varB" = "t1" -o "$varB" = "t2" \) ]
…
```

```
# bash idiomatic way:

if [[ $varA == 1 && ($varB == "t1" || $varC == "t2") ]]; then
```

[[ ]] advanced features

- Regular Expression matching with the =~ operator

```
WORD="something"

[[ $WORD =~ [sS]omething ]]
echo $?
0

[[ $WORD =~ [sS]omethings ]]
echo $?
1
```

Topic

T1.3 – Case Statement

https://youtu.be/JdYZ9JqW4Po

It's the bash's version of a switch statement!

- Compare strings

```
#!/bin/bash

case $1 in
      start)
            echo "Starting the service..."
            echo "done"
            ;;
      stop)
            echo "Stopping the service..."
            echo "done"
            ;;
      *)
            echo "what?!"
            ;;
esac
```

We can also use patterns!

```
#!/bin/bash
shopt -s nocasematch
echo "Enter the name of a month."
read month
case $month in
      February)
            echo "There are 28/29 days in $month.";;
      April | June | September | November)
            echo "There are 30 days in $month.";;
      January | March | May | July | August | October | December)
            echo "There are 31 days in $month.";;
      *)
            echo "Unknown month: $month";;
esac
```

https://phoenixnap.com/kb/bash-case-statement

We can also use globbing patterns! – part #2

```
#!/bin/bash
echo "Enter a character:"
read var
case $var in
       [[:lower:]])
              echo "You entered a lowercase character.";;
       [[:upper:]])
              echo "You entered an uppercase character.";;
       [0-9])
              echo "You entered a digit.";;
       ?)
              echo "You entered a special character.";;
       *)
              echo "You entered multiple characters.";;
esac
```

https://phoenixnap.com/kb/bash-case-statement

Topic

T2.1 – Iterative Statements

While & Until loops

https://youtu.be/D0DWDjXPjME

While loop

- Nothing very deep here
- Note the –n for echo

```
#!/bin/bash

# counting from 0 to 9
N=0
while [ $N -lt 10 ]
do
      echo -n "$N "
      (( N++ ))
done
echo
```

until loop

- Same stuff

```
#!/bin/bash

# counting from 0 to 9
N=0
until [ $N -ge 10]
do
      echo -n "$N "
      (( N++ ))
done
echo
```

Topic

T2.2 – Iterative Statements

For loops

https://youtu.be/afKo5kEZwTA

for loop with a range

- Same example
- Range syntax is inclusive
- Additional syntax to specify a step

```
#!/bin/bash

# counting from 0 to 9
for VALUE in {0..9}
do
      echo -n "$VALUE "
done
echo

# counting from 0 to 9, by increments of 2
for VALUE in {0..9..2}
do
      echo -n "$VALUE "
done
echo
```

for loop with a list of values (as strings)

- Space separates a value from the next

```
#!/bin/bash

NAMES="one two three four five"

for VALUE in $NAMES
do
      echo -n "$VALUE "
done

echo
```

What if we want to use : as separator?

- Space separates a value from the next
- What about the IFS?

"The shell treats each character of $IFS as a delimiter, and splits the results of the other expansions into words using these characters as field terminators."

https://www.gnu.org/software/bash/manual/html_node/Word-Splitting.html

```
#!/bin/bash

NAMES="one:two:three:four:five"

OLDIFS=$IFS
IFS=':'

for VALUE in $NAMES
do
      echo -n "$VALUE "
done

echo

IFS=$OLDIFS
```

for loop over the parameters of the script

- Space separates a value from the next

```
#!/bin/bash

for VALUE in $@
do
      echo -n "$VALUE "
done
echo
```

Interlude: $@ vs. $*

Variable	Meaning
$@	All arguments (as an array)
$*	All arguments (as a single string)

- No differences

HOWEVER…

- Hard to tell if there were iterations at all since we display everything on the same line

```
#!/bin/bash

for VALUE in $@
do
      echo -n "$VALUE "
done
echo

for VALUE in $*
do
      echo -n "$VALUE "
done
echo
```

Interlude: $@ vs. $*

Variable	Meaning
$@	All arguments (as an array)
$*	All arguments (as a single string)

- Still no differences

```
#!/bin/bash

for VALUE in $@
do
      echo "parameter = $VALUE "
done

for VALUE in $*
do
      echo "parameter = $VALUE "
done
```

Interlude: $@ vs. $*

Variable	Meaning
$@	All arguments (as an array)
$*	All arguments (as a single string)

- Differences!!!!

"$@" has the " " applied to each parameter separately: "one" "two" "three" "four" …

"$*" expands into one single parameter: "one two three four…"

```
#!/bin/bash

for VALUE in "$@"
do
      echo "parameter = $VALUE "
done

for VALUE in "$*"
do
      echo "parameter = $VALUE "
done
```

Topic

Bash Functions

https://linuxize.com/post/bash-functions/

The basics

- 2 syntaxes to declare them
- Declare them before to call them

```
#!/bin/bash

# syntax #1 - multi lines
my_function_1a () {
      echo "my_function 1 here"
}
# syntax #1 - single line
my_function_1b () {     echo "my_function 1 here" ; }


# syntax #2 - multi lines
function my_function_2a {
      echo "my_function 1 here"
}
# syntax #2 - single line
function my_function_2a { echo "my_function 1 here" ; }
```

Where to declare / use Functions?

- In scripts (obvious) → see previous slide

BUT ALSO…

- At the command line itself!

```
tux@penguin: my_function () { echo "woot!" ; }
tux@penguin: my_function
woot!
tux@penguin: unset my_function
tux@penguin: my_function
my_function: command not found
tux@penguin:
```

When are functions taken into consideration?

Precedence in execution:

- #1 aliases
- #2 keywords: e.g., function, if, while…
- #3 functions calls
- #4 builtins
- #5 external commands

Variables Scope

- All variables are global
  - This includes those declared INSIDE a function
- Local variables require the `local` keyword
  - They shadow global variables with same name.
- Global variables can be changed from within the function.

```
#!/bin/bash
var1='A'
var2='B'

my_function () {
        var1='C'
        var2='D'
        echo "Inside function: var1: $var1, var2: $var2"
}

echo "Before function: var1: $var1, var2: $var2"
my_function
echo "After function: var1: $var1, var2: $var2"
```

```
Before executing function: var1: A, var2: B
Inside function: var1: C, var2: D
After executing function: var1: C, var2: D
```

Return Value

- Not used the same way as with functions in most programming languages
- The return value is an exit status: that of the last command executed in the function
- Return statement used to set that exit status if needed
- We may inspect it with $?

```
#!/bin/bash

my_function () {
        echo "some result"
        return 55
}

my_function
echo $?
```

```
some result
55
```

So… how do we actually return a value?

- Use a global variable (ewww…)

```
#!/bin/bash

my_function () {
      func_result="some result"
}

my_function
echo $func_result
```

```
some result
```

- Use STDOUT (o.O)

```
#!/bin/bash

my_function () {
      func_result="some result"
       echo "$func_result"
}

func_result="$(my_function)"
echo $func_result
```

```
some result
```

Parameters Passing

- The function is like a script inside the script
- → it uses positional argument variables too!

```
#!/bin/bash

greeting () {
 echo "Hello $1"
}

greeting "Joe"
```

```
Hello Joe
```

Variable	Meaning
$1 $2 $3 … $n	Positional parameters
$0	Function's name
$#	Number of positional parameters passed to the function
$* $@	All the parameters passed to the function. Both are identical when not quoted
"$*"	expands to a single string separated by space (the first character of IFS) - "$1 $2 $n"
"$@"	expands to separate strings - "$1" "$2" "$n"

Comprehensive Example

```
arguments () {
        echo The function location is $0
        echo There are $# arguments
        echo "Argument 1 is $1"
        echo "Argument 2 is $2"
        echo "<$@>" and "<$*>" are the same.
        echo See the difference:
        echo "* gives:"
        for arg in "$*"; do echo "<$arg>"; done
        echo "@ gives:"
        for arg in "$@"; do echo "<$arg>"; done
}

arguments hello world
```

```
The function location is ./arguments.sh
There are 2 arguments
Argument 1 is hello
Argument 2 is world
<hello world> and <hello world> are the same.
See the difference:
* gives:
<hello world>
@ gives:
<hello>
<world>
```