CHALMERS

GÖTEBORGS UNIVERSITET

Introduction to Linux

Lecture 6

Prof. Dr. Philippe Tassin
Chalmers University of Technology

Dr. Evangelos Siminos
Chalmers University of Technology and University of Gothenburg

Göteborg, 8 May 2019

Repetition constructs

Four different repetition constructs:

- for … do … done
- while … do … done
- until … do … done
- select … do … done

for … do … done

```
for NAME [in LIST ]; do COMMANDS; done
```

- `NAME` is the name of a variable that contains the successive elements from the list
- `LIST` is the list of alternatives; can be specified literally or generated with brace expansion, filename expansion, command substitution, etc.

```
[carol@octarine ~/html] cat html2php.sh
#!/bin/bash
# specific conversion script for my html files to php
LIST="$(ls *.html)"
for i in $LIST ; do
     NEWNAME=$(ls "$i" | sed -e 's/html/php/')
     cat beginfile > "$NEWNAME"
     cat "$i" | sed -e '1,25d' | tac | sed -e '1,21d'| tac >> "$NEWNAME"
     cat endfile >> "$NEWNAME"
done
```

while … do … done

- `while CONTROL-COMMAND; do CONSEQUENT-COMMANDS; done`

- `CONTROL-COMMAND` can be any command(s) that exit with a success or failure status (also the [ ] and [[ ]] test statements)

- `CONSEQUENT-COMMANDS` can be any program, script, or shell construct

Test statements

- Different possibilities:
  - [ ]

[FILE1 -ef FILE2]	True if FILE1 and FILE2 refer to the same device and inode numbers.
[-o OPTIONNAME]	True if shell option "OPTIONNAME" is enabled.
[-z STRING]	True of the length if "STRING" is zero.
[-n STRING] or [STRING]	True if the length of "STRING" is non-zero.
[STRING1 == STRING2]	True if the strings are equal. "=" may be used instead of "==" for strict POSIX compliance.
[STRING1 != STRING2]	True if the strings are not equal.

    [ ] is a POSIX standard
  - `test` command: this is the same thing as [ ]
  - [[ ]] works in Bash and a few other shell, but not POSIX; has some additional features
  - (( )) arithmetic expansion:

    exit code is zero (“true”) if result is nonzero! Not POSIX
  - `command`: exit code of command will be used

until … do … done

```
until TEST-COMMAND; do CONSEQUENT-COMMANDS; done
```

select … do … done

- This is almost the same as a for loop, but the list alternatives are printed as numbered alternatives
- `select WORD [in LIST]; do RESPECTIVE-COMMANDS; done`
- is used to create menus in interactive scripts

```
[carol@octarine testdir] cat private.sh
#!/bin/bash

echo "This script can make any of the files in this directory private."
echo "Enter the number of the file you want to protect:"

select FILENAME in *;
do
     echo "You picked $FILENAME ($REPLY), it is now only accessible to you."
     chmod go-rwx "$FILENAME"
done

[carol@octarine testdir] ./private.sh
This script can make any of the files in this directory private.
Enter the number of the file you want to protect:
1) archive-20030129
2) bash
3) private.sh
#? 1
```

Break / continue

- `break` exits the repetition construct before its normal ending

- `continue` steps to the next iteration in the repetition construct
  - in a for loop: the next value in `LIST` is taken
  - in a while or until loop: the `TEST-COMMAND` is checked again

Positional arguments in repetitive constructs

- If `LIST` is not specified it is by default the list of positional arguments `$@`

- `shift N` built-in command removes first `N` positional arguments

  → useful for parsing the arguments of a script or function

Functions

- Functions can be used to group commands
- function FUNCTIONNAME { COMMANDS; }
  or
  FUNCTIONNAME () { COMMANDS; }

  (note the spaces around the curly braces)
- Functions are like little scripts and can have variables and positional arguments
- Functions in Bash don't have a return value. If data must be returned to the calling command, a shell or environment variable must be used
- `return` can be used to terminate the function and return an exit code

Variables, positional parameters in functions, and exit status

- `local variablename`: this creates a variable local to the function
- The positional parameters `$1`, `$2`, etc. are the arguments to the function (not the arguments to the script containing the function)
- The function can be quit (and execution be returned to the command after the function call) with `return`
- `return x` can also be used to return an exit status to the calling process

Variables in Bash

- Variables is Bash don't need to be declared

- But they can be explicitly declared:

```
declare OPTIONS VARIABLENAME
```

- Mostly useful for integer type variables; arithmetic expansion is automatically performed if variables are declared as integers (although I recommend using (( )) expansion:

```
[tassin@hebbe ~]$ VAR=10+2
[tassin@hebbe ~]$ echo ${VAR}
10+2
[tassin@hebbe ~]$ declare -i VAR=10+2
[tassin@hebbe ~]$ echo ${VAR}
12
```

- Some other OPTIONS exist; see the textbook for more information

Constants

- Constants = read-only variables
- `readonly OPTION VARIABLE(s)`

Arrays

- You can use arrays in Bash; indices are zero based

- `declare -a ARRAYNAME`
- `ARRAYNAME=(value1 value2 ... valueN)`
- `ARRAYNAME[indexnumber]=value`

Dereferencing array elements

```
[bob in ~] ARRAY=(one two three)

[bob in ~] echo ${ARRAY[*]}
one two three

[bob in ~] echo $ARRAY[*]
one[*]

[bob in ~] echo ${ARRAY[2]}
three

[bob in ~] ARRAY[3]=four

[bob in ~] echo ${ARRAY[*]}
one two three four
```

Any other questions?