Lecture 4: Loops and few other thingies

Last update: 20220505

Table of Contents

1. Scripts vs. functions

Now that we have seen how to implement in **Bash** both scripts and functions, we can discuss briefly their similarities, differences and typical use cases.

First, let us start with the execution details of scripts. In general, we run any **Bash** script either by 'sourcing' or by 'executing' that script. The first case corresponds to the following syntax:

```
source someScript.sh # sourcing the script
```

When executed this way, all lines in the script are read and executed by **Bash** one-by-one, just as if they were typed separately line by line in the terminal. The sourced script inherits the environment from the terminal (i.e. from the current shell), and can modify it globally. The exit status of script must be specified with the keyword **return**. Script does not run in a separate process (more on this later).

```
someScript # executing the script
```

This way, you are running your script as any other **Linux** or **Bash** command. As we already saw, this will work only if the directory where the file with the source code of script sits was added to the environment variable **PATH**, and if that file has also the execute (`x`) permission. The executed script does not inherit by default the environment from the terminal (only variables, functions, etc., which were defined with **export** are inherited), and cannot modify it globally. Therefore, it is much safer to run scripts this way, if you want to keep your current shell environment clean. The exit status of the executed script is specified with the keyword **exit**. When executed this way, the script runs in a separate process (more on this later).

If you do not want to make the script executable by adding to it (`x`) permission, you can always run the shell explicitly and tell it to process the file like it was an executable, with the following syntax:

```
bash someScript.sh # executing the non-executable script
```

On the other hand, functions behave differently. After you source the file where a function is implemented, **Bash** stores that function in the computer's memory, and from that point onwards you can use that function as any other **Linux** or **Bash** command. For functions, there is no need to bother with using keyword **source**, setting the execute permission, modifying **PATH**, etc. That means that if you have added to your `~/.bashrc` the following line:

```
source ~/functions.sh
```

where in the example file `~/functions.sh` you have the implementation of your **Bash** functions, you can use effortlessly all your functions in any new terminal you open.

Functions are much more suitable for making long scripts modular. In terms of environment protection, functions are much cleaner to use than scripts, due to keyword **local**, which can be used only in the function body, and which limits the scope and lifetime of a variable defined in the function only to the function execution.

If a function **someFunction** and a script **someScript** with execute permission have exactly the same implementation, then executing in the terminal **someFunction** only by its name is more efficient than executing in the terminal a script **someScript** only by its name, because **Bash** function does not start a separate process.

Programmatically, you can fetch the function name in its body implementation via built-in variable **FUNCNAME** (typically by having **echo $FUNCNAME** at the beginning of function body). For scripts, the file name in which the script was implemented can be obtained programmatically from the built-in variable **BASH_SOURCE**. This becomes very important when inspecting only the printout of your code execution (e.g. for debugging purposes), when it is easy to trace back which function or script produced which part of the final result (in this context, the built-in variable **LINENO** can also be handy, because **echo $LINENO** prints literally the line number of the source code where this variable is referenced).

We summarize the above thorough comparison with the following final conclusion: Use **Bash** scripts only for the very simple cases and **Bash** functions for everything else.

2. Command chain: && and ||

Since every command in **Linux** and **Bash** has the exit status, it is possible programmatically to branch the code execution, depending on whether a command has executed successfully (exit status 0), or has failed during execution with some error status (exit status 1.. 255). For instance, we would like multiple commands to execute one after another, but only if all of them executed successfully. As soon as one command has failed, we would like immediately to abort the execution of all subsequent commands. In **Bash**, we can achieve that with the *command chain*.

The command chain is a sequence of commands separated either with `&&` or `||` operators. If two commands are chained by `&&`, the second command will be executed only if the first one executed successfully. For instance:

```
$ mkdir someDirectory && echo "New directory was made."
New directory was made.
```

You will see the printout from **echo** only if the directory was successfully made with the command **mkdir**. On the other hand, if **mkdir** has failed, the command chain has broken, and **echo** is not executed. For instance, we can intentionally mistype **mkdir** just to simulate the failure of the first command in the chain:

```
$ mkdirrr someDirectory && echo "New directory was made."
mkdirrr: command not found
```

In this case, **echo** is not executed because the failure of **mkdirrr** has broken the command chain `&&`.

If two commands are chained by `||` operator, the second command in the chain will be executed only if the first command has failed:

```
$ mkdirrr someDirectory || echo "Cannot make directory. Sorry."
mkdirrr: command not found
Cannot make directory. Sorry.
```

The frequent use case of the command chain is to combine both `&&` and `||` operators in the following way:

1. start a command chain by grouping multiple commands with the `&&` operator;
2. append at the end of command chain the very last command with the `||` operator.

Schematically:

```
command1 && command2 && command3 ... || lastCommand
```

The main point behind this construct is the following: **lastCommand** is executed if and only if any of the commands **command1**, **command2**, ..., has failed. The command **lastCommand** is not executed only if all of the commands **command1**, **command2**, ..., have executed successfully. Typically, the last command in the above chain would be some error printout accompanied by the code termination, either with **exit** or **return**. Therefore, the **lastCommand** is a sort of safeguard for the execution of all previous commands in the chain.

Example: Consider the following command chain

```
echo "Hello" && pwd && date || echo "Failed"
```

Since all commands executed successfully, it creates the following output:

```
Hello
/home/abilandz/Lecture/PH8214/Lecture_4
Mi 15. Mai 07:53:25 CEST 2019
```

The very last command after `||` operator, **echo "Failed"**, is not executed. Now we introduce some error, e.g. we mistype something intentionally:

```
echo "Hello" && pwddd && date || echo "Failed"
```

Now the output is:

```
Hello
pwddd: command not found
Failed
```

The first command in the `&&` chain executed successfully, and the execution continued with the next command in the `&&` chain. However, the second command **pwddd** has failed, and therefore has broken the `&&` chain. From that point onwards, only the command after `||` will be executed, and all the remaining commands in `&&` chain are ignored (the command **date** in this case).

In practice, the most frequent use case of the command chain in sourced scripts or in functions is illustrated schematically:

```
someCommand || return 1
someOtherCommand || return 2
...
```

or in executables as

```
someCommand || exit 1
someOtherCommand || exit 2
...
```

This way, it is possible to add easily an additional layer of protection for the execution of any command in your **Bash** code. Moreover, since the exit status is stored in the special variable **$?**, it is also possible by inspecting its content upon termination, to fix programmatically the particular reason of the failure, without intervening manually in the code.

3. Test construct: [[...]]

For simple testing in **Bash**, we can use either `[[ ... ]]` or `[ ... ]` constructs. The construct `[[ ... ]]` is more powerful than `[ ... ]` since it supports more operators, but it was added to **Bash** later than `[ ... ]`, meaning that it will not work with some older **Bash** versions. There are corner cases where their behavior differs, since their implementation is conceptually different:

```
$ type [[
[[ is a shell keyword
$ type [
[ is a shell builtin
```

For instance, the quotes can be omitted inside `[[` but not inside `[`. But in most cases of practical interest, `[[ ... ]]` and `[ ... ]` behave in the same way and yield the same results.

Test constructs also return the exit status — if the test was successful the exit status is set to 0 also in this context. Which operators we can use within these two test constructs depends on the nature of the content of the variable(s) we are putting to the test. Roughly, we can divide the use case of the test construct `[[ ... ]]` in the following 3 categories, and we enlist the meaningful operators for each category:

- General case: `-z, -n, ==, != , =~`
- Integers: `-gt, -ge, -lt, -le, -eq`
- Files and directories: `-f, -d, -e, -s, -nt, -ot`

These 3 distinct categories of the usage of `[[ ... ]]` are best explained with a few concrete examples — we start with the general case.

General case

Example 1: How to check if variable **Var** has been initialized?

```
[[ -n ${Var} ]] && echo Yes || echo No
```

Remember the correct syntax and the extreme importance of empty characters within the test construct `[[ ... ]]`, as this is a typical source of errors:

```
[[ -n ${Var} ]] # correct
[[-n ${Var} ]]  # wrong
[[ -n ${Var}]]  # wrong
[[ -n${Var} ]]  # wrong
```

The very frequent use case is to check at the very beginning of the body of a script or a function if the user has supplied some value for the mandatory argument:

```
[[ -n ${1} ]] || return 1
```

If the user didn't provide value for the first argument, the above code snippet will terminate the subsequent execution.

The operator `-n` accepts only one argument and checks whether it is set to some value, the opposite is achieved with `-z` which exits with 0 if its argument is not set.

If you forgot to specify an operator within the test construct, it is defaulted to `-n`, i.e.

```
[[ ${Var} ]]
```

is equivalent to

```
[[ -n ${Var} ]]
```

Example 2: How to check if the content of variable **Var1** is equal to the content of variable **Var2**?

We can illustrate this example with the following code snippet:

```
Var1=a
Var2=ab
[[ ${Var1} == ${Var2} ]] && echo Yes || echo No
```

Note that `==` is the comparison operator, while `=` is the assignment operator. The comparison operator `==` expects two arguments, and it treats both LHS and RHS arguments as strings. Since by default any variable in **Bash** is a string, this operator is applicable to any variable content. In particular, you can also compare integers this way, but it's much safer to do integer comparison with the `-eq` operator, as explained below. The operator `!=` does the opposite to `==`, i.e. it exits with 0 if two strings are not the same.

Example 3: How to check if one string contains another string as a substring?

```
Var1=abcd
Var2=bc
[[ ${Var1} =~ ${Var2} ]] && echo "Var1 contains Var2"
```

This frequently used operator is supported only within `[[ ... ]]`, but not within `[ ... ]`.

The executive summary for the first category of operators is provided with the following table:

Operator	Outcome (exit status)
[[-z ${Var}]]	true (0) if Var is zero (null)
[[-n ${Var}]]	true (0) if Var holds some value
[[${Var1} == ${Var2}]]	true (0) if Var1 and Var2 are exactly the same
[[${Var1} != ${Var2}]]	true (0) if Var1 and Var2 are not exactly the same
[[${Var1} =~ ${Var2}]]	true (0) if Var1 contains Var2 as a substring

Integers

When it comes to the second group of operators, `-gt, -ge, -lt, -le, -eq`, they are specific in a sense that they can accept only integers as arguments.

Example 4: How to check if one integer is greater than some other integer?

```
Var=44
[[ ${Var} -gt 10 ]] && echo Yes || echo No
```

Quite frequently, if your script or function demands that a user must provide exactly the certain number of arguments, you can use the following standard code snippet at the beginning of your code:

```
[[ $# -eq 2 ]] || return 1
```

In the above example, if a user did not provide exactly two arguments, the code execution terminates. It is always safer to compare two integers with `-eq` than to treat them as strings with `==`, due to corner cases like this one:

```
[[ 1 == 01 ]] && echo Yes || echo No # prints No
[[ 1 -eq 01 ]] && echo Yes || echo No # prints Yes
```

As a side remark, we indicate that prepending '0' to a number is not trivial, and in fact, that is a widely accepted convention in a lot of programming languages to change the representation of a number from decimal (default) into an octal base. Therefore, this doesn't work:

```
$ [[ 8 -eq 08 ]] && echo Yes || echo No
bash: [[: 08: value too great for base (error token is "08")
No
```

Since the meaning of integer operators is rather obvious, we just provide the executive summary of their usage with the following table:

Operator	Outcome (exit status)
[[${Var1} -gt ${Var2}]]	true (0) if Var1 is greater than Var2
[[${Var1} -ge ${Var2}]]	true (0) if Var1 is greater than or equal to Var2
[[${Var1} -lt ${Var2}]]	true (0) if Var1 is smaller than Var2
[[${Var1} -le ${Var2}]]	true (0) if Var1 is smaller than or equal to Var2
[[${Var1} -eq ${Var2}]]	true (0) if Var1 is equal to Var2

Files and directories

The very last group of operators, `-f, -d, -e, -s, -nt, -ot`, expects their argument(s) to be either files or directories. The first four accept one argument, while the last two take two arguments. Their meaning is illustrated in the following examples.

Example 5: How to check if the file `${HOME}/test.txt` exists or not?

```
Var=${HOME}/test.txt
[[ -f ${Var} ]] && echo "${Var} exists." || echo "${Var} doesn't exist."
```

Analogously, we can check for the existence of a directory with operator `-d`, for instance:

```
Var=${HOME}/SomeDirectory
[[ -d ${Var} ]] && echo "${Var} exists." || echo "${Var} doesn't exist."
```

Frequently, we want to trigger some code execution only if the file is non-empty, we can check that with the operator `-s`, as in the following example:

```
Var=${HOME}/test.txt
[[ -s ${Var} ]] && echo "${Var} is not empty" || echo "${Var} is empty"
```

For instance, if your script or function is expected to extract some data from the file that user needs to supply as the very first argument, you can implement the following protection at the very beginning against the empty file:

```
[[ -s ${1} ]] || return 1
```

Finally, it is possible to compare directly some specific attributes of file metadata, for instance the modification time.

Example 6: How to check if the file `${HOME}/test1.txt` is newer (i.e. modified more recently) than the file `${HOME}/test2.txt`?

This can be answered with operator `-nt` ('newer than') which takes two arguments:

```
File1=${HOME}/test1.txt
File2=${HOME}/test2.txt
[[ ${File1} -nt ${File2} ]] && echo "${File1} is newer" || echo "${File2} is
newer"
```

The executive summary of the most important test operators in this last category is provided in the following table:

Operator	Outcome (exit status)
[[-f ${Var}]]	true (0) if Var is the existing file
[[-d ${Var}]]	true (0) if Var is the existing directory
[[-e ${Var}]]	true (0) if Var is existing file or directory
[[-s ${Var}]]	true (0) if Var is a file, and is not empty
[[${Var1} -nt ${Var2}]]	true (0) if a file Var1 is newer than a file Var2
[[${Var1} -ot ${Var2}]]	true (0) if a file Var1 is older than a file Var2

When it makes sense and it is convenient, it is possible to refine further the above examples with the negation operator `!`, for instance:

```
[[ ! -f ${Var} ]] # true (0) if Var is NOT the existing file
```

In this section we have summarized the most important options — for the other available options, check the corresponding documentation of test constructs by executing in the terminal:

```
help test
```

In the end, we indicate that the test construct `[[ ... ]]` can be used to branch the code execution, depending on whether some command executed correctly, or it has failed. If it has failed, we can branch even further the code execution depending on the exit status of a particular error. This is achieved by storing and testing the content of special variable **$?**, schematically:

```
someCommand # variable $? gets updated with the exit status of this command
ExitStatus=$? # store permanently the exit status of previous command in this
variable
[[ ${ExitStatus} -eq 0 ]] && some-code-if-command-worked
[[ ${ExitStatus} -eq 1 ]] && some-other-code-to-handle-this-particular-error-
state
[[ ${ExitStatus} -eq 2 ]] && some-other-code-to-handle-this-particular-error-
state
...
```

Later we will see that such a code branching can be optimized even further with `if-elif-else-fi` or `case-in-esac` command blocks.

4. Catching user input: read

We have seen already how variables can be initialized in a non-interactive way, by initializing them with some concrete values at declaration. Now we discuss how the user's input from the keyboard can be on-the-fly stored directly in some variable. In essence, this feature enables **Bash** scripts and functions to be interactive, in a sense that during the code execution (i.e. at *runtime*),

with your input from the keyboard you can steer the code execution in one direction or another. This is achieved with a very powerful **Bash** built-in command **read**.

By default, the command **read** saves input from the keyboard into its own variable **REPLY**. Alternatively, you can specify yourself directly the name of the variable(s) which will store the input from the keyboard. This is best illustrated with examples.

Example 1: If we use **read** without arguments, the entire line of user input is stored in the variable **REPLY**, as this code snippet demonstrates:

```
read
```

After you have executed **read** in the terminal, this command is waiting for your input from the keyboard. Just type some example input, e.g. `1 22 abc`, and press 'Enter'. Now you can programmatically retrieve that input:

```
$ echo ${REPLY}
1 22 abc
```

Instead of relying on variable **REPLY**, another generic usage of **read** is to specify one or more arguments explicitly, in the following schematic way:

```
read Var1 Var2 ...
```

This version takes a line from the keyboard input and breaks it down into words delimited by input field separators. The default input field separator is an empty character, and the input is terminated by pressing the 'Enter'.

Example 2: The previous example re-visited, but now using **read** with arguments.

```
read Name Surname
```

After typing that in the terminal, **read** is waiting for your feedback. Type something back, e.g. `James Hetfield`, and press 'Enter'. Now type in the terminal:

```
$ echo "Your name is ${Name}."
Your name is James.
$ echo "Your surname is ${Surname}."
Your surname is Hetfield.
```

The user-supplied arguments to the **read** command, **Name** and **Surname**, have become variables **Name** and **Surname**, initialized with the user's input from the keyboard, `James` and `Hetfield`, respectively.

If there are more words in the user's input from the keyboard than the variables supplied as arguments to **read**, all excess words are stored in the last variable.

Example 3: The previous example re-re-visited, but now using **read** with fewer arguments that there are words in the user's input.

```
read Var1 Var2
```

Feed to **read** the following input from the keyboard `1 22 a bb`, and press 'Enter'. If you now execute in the terminal

```
echo "Var1 is ${Var1}"
echo "Var2 is ${Var2}"
```

for the output you get:

```
Var1 is 1
Var2 is 22 a bb
```

The branching of the code execution at runtime, depending on the user's input from the keyboard, can be achieved in the following simplified and schematic way:

```
read Answer
[[ ${Answer} == yes ]] && do-something-if-yes
[[ ${Answer} == no ]] && do-something-if-no
```

In combination with `if-elif-else-fi` and `case-in-esac` statements (to be covered later!) the **read** command offers a lot of flexibility on how to handle and modify the code execution at runtime.

The default behavior of **read** can be modified with a bunch of options (check **help read** for the full list). Here we summarize only the ones which are used most frequently:

```
-p : specify prompt
-s : no printing of input coming from a terminal
-t : timeout
```

For instance:

```
read -p "Waiting for the answer: "
echo ${REPLY}
```

The specified message in the prompt of **read** can hint to the user what to type as an answer:

```
read -p "Please choose either 1, 2 or 3: "
echo ${REPLY}
```

For more complicated menus, **Bash** offers built-in command **select** which is covered later in the lecture.

The flag `-s` ('silent') hides in the terminal the user input:

```
read -s Password
```

Now the user's input is not showed in the terminal as you typed it, but it was stored nevertheless in the variable **Password**. Within your subsequent code you can programmatically do some checks on the content of **Password**. If you remove the read permission on that file in which you are doing those checks, you have obtained a very simple-minded mechanism to handle passwords, etc.

Finally, with the following example:

```
read -t 5
```

the user is given 5 seconds to provide some input from a keyboard. If the user within the specified time interval does not provide any input, the **read** command reaches the timeout and terminates. The code execution proceeds like nothing happened. Therefore, within the specified time interval we are given the chance to type something and to modify the default execution of the code. All the above flags can be combined, which can make the usage of **read** command quite handy, and scripts both interactive and flexible during execution.

The command **read** can be used in some other contexts as well, e.g. to parse the file content line-by-line in combination with the **while** loop — this is covered at the end of today's lecture.

5. Arithmetic in Bash

We have already seen that, whatever is typed first in the terminal and before the next empty character is encountered, **Bash** will try to interpreted as command, function, etc. For this reason, we cannot do directly arithmetic in **Bash**. For instance:

```
$ 1+1
1+1: command not found
```

is producing an error, because a command named **1+1** doesn't exist. Other trials produce slightly different error messages, but the reason for the failure is aways the same:

```
$ 1+ 1
1+: command not found
$ 1 + 1
1: command not found
```

Instead, we must use the special operator `(( ... ))` to do integer arithmetic in **Bash**. For instance:

```
echo $((1+1))
```

produces the desired printout

```
2
```

The operator `(( ... ))` can also swallow the variables:

```
Counter=1
((Counter+=10)) # doing some integer manipulation
echo ${Counter} # prints 11
```

Within `(( ... ))` we can use all standard operators to perform integer arithmetic: `+, -, /, *, %, ++, --, **, +=, -=, /=, *=` , with the self-explanatory meanings.

Example: How to calculate powers of integers in **Bash**? We can raise an integer to some exponent in the following way:

```
Int=5
Exp=2
echo $((Int**Exp)) # prints 25
```

As you can see from the above example, it is not necessary within `(( ... ))` to reference the content of variable explicitly with **$**, the operator itself takes care of that. The following alternatives with lengthier code are also correct:

```
echo $(($Int**$Exp)) # prints 25
echo $((${Int}**${Exp})) # prints 25
```

but clearly it is not as clear and elegant as the very first version.

Operator `(( ... ))` can handle only integers, both in terms of input and output. An attempt to use floating point numbers leads to an error:

```
$ echo $((1+2.4))
bash: 1+2.4: syntax error: invalid arithmetic operator (error token is ".4")
```

Floating point arithmetic cannot be done directly in **Bash**, but this is not a severe limitation, because we can always invoke some **Linux** command to perform it, like **bc** ('basic calculator'), which is always available — more on this later!

When it comes to division which does not yield as the final result an integer, **Bash** does not report the error, instead it reports as the result the integer after the fractional part (remainder) is discarded:

```
echo $((7/3)) # prints 2
echo $((8/3)) # prints 2
echo $((9/3)) # prints 3
```

To get the remainder after the division of two integers, we can use the modulo operator (`%`):

```
echo $((7%3)) # prints 1
echo $((8%3)) # prints 2
echo $((9%3)) # prints 0
```

Besides supporting integer arithmetic operators within, `(( ... ))` we can also perform integer comparison by using the familiar `<`, `<=`, `==`, `!=`, `>=` and `>` operators. This is an alternative to integer comparison within the test construct `[[ ... ]]` which has its own operators for integer comparison. For instance, the following code snippet

```
(( ${Var1} < ${Var2} ))
```

is equivalent to

```
[[ ${Var1} -lt ${Var2} ]]
```

and so on.

We now highlight the following common mistake: The meaning of operator `+=` within and outside of `(( ... ))` is different. That is illustrated with the following examples:

```
NumberOfWords=0
echo $NumberOfWords # prints 0
NumberOfWords+=1
echo $NumberOfWords # prints 01
NumberOfWords+=1
echo $NumberOfWords # prints 011
```

When used outside of `(( ... ))`, the operator `+=` is just a shorthand operator to combine strings. On the other hand:

```
NumberOfWords=0
echo $NumberOfWords # prints 0
((NumberOfWords+=1))
echo $NumberOfWords # prints 1
((NumberOfWords+=1))
echo $NumberOfWords # prints 2
```

The most frequent use case of `(( ... ))` operator is to increment the content of variable within loops, which we cover next.

6. Loops: for, while and until

Just like any other programming language **Bash** also supports loops. The most frequently used loops are **for** and **while** loops, and only they will be discussed in this section in detail. The third possibility, the loop **until**, differs only marginally from the **while** loop, and therefore it will not be addressed separately. In particular, the **while** loop runs the loop *while* the condition is `true`, where the **until** loop runs the loop *until* the condition is `true` (i.e. while the condition is `false`). Besides that, there is no much of a difference between these two versions, and it is a matter of personal taste which one is preferred in practice. On the other hand, there are a few non-trivial differences between **for** and **while** loops, both in terms of syntax and use cases.

The syntax of **for** and **while** loops is fairly straightforward, and can be grasped easily from a few concrete examples. We start first with the examples for the **for** loop.

Example 1: Looping over the specified list of elements.

```
for Var in 1 2 3 4; do
 echo "$Var"
done
```

The output is:

```
1
2
3
4
```

This version of **for** loop iterates over all elements of a list. These elements are specified between the keyword **in** and delimiter `;`. If you omit `;` the list needs to be terminated with the new line. Therefore, a completely equivalent implementation is:

```
for Var in 1 2 3 4
do
 echo "$Var"
done
```

Elements of a list are separated with the empty characters, and elements can be pretty much anything, e.g. consider:

```
Test=abc
for Var in 1 ${Test} 4.44; do
 echo "$Var"
done
```

The output is:

```
1
abc
4.44
```

Later we will see that we can even loop directly over the output of some command (e.g. over all files in a certain directory which match some naming convention, etc.).

Example 2: Looping over all arguments supplied to a script or a function.

We have already seen that we can loop over all arguments supplied to a script or a function in the following way:

```
for Arg in "$@"; do
 echo "Argument is: ${Arg}"
done
```

Since this is a frequently used feature, there exists a shorthand version when you need to loop over the arguments. Consider the following script named `forLoop.sh`, in which we have dropped completely the list of elements in the first line of **for** loop:

```
#!/bin/bash

for Arg; do
 echo "Argument is: $Arg"
done

return 0
```

By executing

```
source forLoop.sh a bb ccc
```

you get the following printout:

```
Argument is: a
Argument is: bb
Argument is: ccc
```

Therefore, if the list of elements is not explicitly specified in the first line of **for** loop, the list of elements has been defaulted to all arguments supplied to that script or function.

There exists also the C-style version of **for** loop in **Bash**, which can handle explicitly the increment of a variable. The C-style version looks schematically as:

```
MaxValue=someValue
for ((Counter=0; Counter<$MaxValue; Counter++)); do
 ... some commands ...
done
```

When it comes to the **while** loop, it is used very frequently and conveniently in combination with the test construct `[[ ... ]]`. The following code snippets illustrate its most typical use cases. For the C-style **while** loop, we would use the following example syntax:

```
Counter=1
while [[ $Counter -lt 10 ]]; do
 echo "Counter is equal to: $Counter"
 ((Counter++))
done
```

Another frequently used case is illustrated in the following example:

```
while [[ -f someFile ]]; do # check if the file exists
 ... some work involving the file someFile ...
 sleep 1m # pause code execution for 1 minute
done
```

This loop will keep repeating as long as the file `someFile` is available. When the file is deleted, `[[ -f someFile ]]` evaluates to `false`, and the loop terminates.

As a side remark, in the above example we have used the trivial, nevertheless sometimes very handy, **Linux** command **sleep**. This command does nothing, except that it delays the code execution for the time interval specified via the argument. The argument can be interpreted as the time interval either in seconds (s), minutes (m), hours (h) or days (d):

```
sleep 10m # pause the code execution for 10 minutes
sleep 2h  # pause the code execution for 2 hours
```

This command can be used in some simple-minded cases to avoid a conflict among concurrently running processes. Another use case is to determine the periodicity of infinite loops.

Example 3: Infinite loops with the specified periodicity.

The following loop will keep running forever, with the periodicity of once per hour:

```
while true; do
 ... some code that you need again and again ...
 sleep 1h
done
```

In the above code snippet, we have used the **Bash** built-in command **true**, which does nothing except it returns the success exit status 0 each time it is called. There is also **Bash** built-in command **false**, which does nothing except it returns the error exit status 1.

A more sophisticated way to set up the scheduled execution of your code can be achieved with the command **crontab** (check its man page).

With the keywords **continue** and **break** you can either continue or bail out from **for**, **while** and **until** loops. Outside of these three loops these commands are meaningless, and will produce an error. Their usage is illustrated with the following code snippet:

```
Counter=0
Max=4
while true; do
 ((Counter++))
 [[ ${Counter} -lt ${Max} ]] && echo "Still running" && continue
 echo "Terminating" && break
done
```

Upon execution, it leads to the following printout:

```
Still running
Still running
Still running
Terminating
```

If you have nested the loops, you can from the inner loop continue or break directly the outer loop. The level of the outer loop that you want to continue or break, is specified with the following syntax:

```
break someInteger
```

or

```
continue someInteger
```

In the next section, we discuss how we can combine some of these different functionalities, and establish another frequently used feature, which is especially handy when we need to parse through the file content line-by-line.

7. Parsing the file content: while+read

Very frequently, we need within a script or a function to parse through the content of an external file, and to perform some programmatic action line-by-line. This can be achieved very conveniently by combining the **while** loop and the **read** command. We remark, however, that this is not the most efficient way to parse the file content, its usage is recommended only for the short files.

As a concrete example, let us have a look at the following script, named `parseFile.sh`. This script takes one argument and that argument must be a file:

```
#!/bin/bash

File=$1 && [[ -f $File ]] || return 1

while read Line; do
 echo "I am reading now: $Line"
 sleep 1s
done < $File

return 0
```

The content of the file is redirected to the loop with `<` operator at the end of the loop.

Then, edit some temporary file, named for instance `data.log`, with the following simple content:

```
10 20 30
100 200
abcd
```

Finally, execute the script with:

```
source parseFile.sh data.log
```

The printout in the terminal is:

```
I am reading now: 10 20 30
I am reading now: 100 200
I am reading now: abcd
```

As we can see, **while+read** construct automatically reads through all the lines in the file, and in each iteration the whole content of the current line is stored in the variable which we have passed as an argument to the **read** command (in the above example it is the variable named **Line** — if we do not specify any variable, then the variable **REPLY** of command **read** is used automatically). That means that in each iteration within the **while** loop we have at our disposal the content of a line from the external file in the variable, and then we can manipulate its content within the script programmatically.