Bash course

T&S - Cheurte

June 19, 2024

Contents

1 Introduction

Bash is a scripting language that can be executed by the bash program. It allows performing a series of actions, such as navigating in specific directories, creating folders or launching processes. By saving these command lines in a script, it is possible to repeat the same sequence of steps multiple times and execute them by running the script.

1.1 Bash commands

Usually, the shell prompt looks something like this:

```
[username@host ~]$
```

It is then possible to enter a command after the `$` sign, and check the output in the terminal. Generally, commands follow this syntax:

```
command [OPTIONS] arguments
```

Some simple commands:

- `echo` command

  The `echo` command, which we used massively already, allow a user to print something on the terminal. However, the echo command has also a few parameters:

  Options of `echo`:

Option	Description
-n	Do not output a trailing newline
-E	Disable the interpretation of the following backslash-escaped characters.
-e	Interpretation of the following backslash-escaped characters in each String

  Backslash-escaped characters list

Option	Description
\a	Alarm sound
\b	One character back
\c	Suppress text wrapping
\f	Back
\n	Line break
\r	Back to beginning of line
\t	Tabulator (horizontal)
\v	Tabulator (vertical)
\\	Backslash character output
\nnn	ACSII characters in octal form (bash only)
\0nnn	ASCII characters in octal form (sh and ksk only)

  It is also possible to specify text attribute on the `echo` command, like colors or properties.

- `date`: Display the current date (There is an extensive documentation on the `date` command, it will not be described here.) `Usage:  date [OPTION]...  [+FORMAT]`

```
[username@host ~]$ date
Tue Jun 11 13:10:03 CEST 2024
```

- `pwd`: Displays the **P**resent **W**orking **D**irectory

```
[username@host ~]$ pwd
/home/user/Documents/
```

Arg.	**Comment**
-L	print the value of $PWD if it names the current working directory (enabled by default)
-P	print the physical directory, without any symbolic links (enabled by default)

- `ls` : **L**ist the content of the current Directory

```
[username@host ~]$ ls
Documents Downloads
```

⚠. A lot of other parameters are available for the `ls` command. Check the doc for more information.

Arg.	**Arg.**	**Comment**
`-a`	`--all`	Do not ignore entries starting with `.`
`-l`		Use a long listing format

- `ps` : displays information about a selection of the active processes:

```
[username@host ~]$ ps
  PID TTY          TIME CMD
 1411 pts/4    00:00:02 bash
11958 pts/4    00:00:00 ps
```

⚠. There is an extensive doc with more parameter with `ps --help all`

- `which` : Print the location of the executable. The `which` command has also a bunch of options.

- The `cat` command concatenate File(s) to standard output:
`Usage: cat [OPTION]... [FILE]...`

Example

```
[username@host ~]$ cat main.c
#include<stdio.h>
int main(){
    printf("Hello World!");
    return 0;
}
```

`cat` **config** (⚠. read the doc for more)

`-T`	`--show-tabs`	Display TAB characters as `Î`
`-E`	`--show-ends`	Display `$` at the end of each line
`-n`	`--number`	Number all output lines

So it is simply possible to get the entire content of a file like this for example:

```
file_content=$(cat file.md)
```

- `wc` print newline, word, and byte counts for each FILE, and a total line if more than one FILE is specified. A word is a non-zero-length sequence of printable characters delimited by white space. `Usage: wc [OPTION]... [FILE]...`

Example

```
[username@host ~]$ cat file.txt
Tue Jun 11 13:10:03 CEST 2024
/home/user/Documents/
[username@host ~]$ wc -lc file.txt
2 52 spec.txt
```

`wc` **Config** (read the doc for more)

Arg.	**Arg.**	**Comment**
-c	––bytes	print the byte counts
-m	––chars	print the character counts
-l	––lines	print the newline counts

- The `grep` search for **PATTERNS** in each file or string.

```
[username@host ~]$ cat main.c
#include<stdio.h>
int main(){
    printf("Hello World!");
    return 0;
}
```

```
[username@host ~]$ grep -i "Hello World" main.c
printf("Hello World!\n");
```

The `-i` is the `--ignore-case` argument is the ignore case distinctions in patterns and data. (⚠. There is again here an extensive manual for this command.)

- `Read` command

  The `read` command is used to read the user input and store it in a variable.

  ```
  echo "Hello, what's your name:"
  read name
  echo Nice to meet you $name
  ```

  ```
  Hello, what's your name:
  Peter
  Nice to meet you Peter
  ```

  - Split the user input

    ```
    read var1 var2
    echo var1: $var1
    echo var2: $var2
    ```

    ```
    Hello World
    var1: Hello
    var2: World
    ```

  Note: The `-a` parameter is selected by default.

Read Options

Options	Description
-a	Assigns the provided word sequence to a variable named
-d	Reads a line until the provided value is typed instead of a new line.
-e	Starts an interactive shell session to obtain the line to read.
-i	Adds initial text before reading a line as a prefix.
-n	Returns after reading the specified number of characters while honoring the delimiter to terminate early.
-N	Returns after reading the specified number of chars, ignoring the delimiter.
-p	Outputs the prompt string before reading user input.
-r	Disable backslashes to escape characters.
-s	Does not echo the user's input.
-t	The command times out after the specified time in seconds.
-u	Read from file descriptor instead of standard input.

⚠. It is possible to get the manual of a command with the `man` command.

2 Bash scripts

2.1 Script name

By naming convention, bash scripts end with `.sh`. However, bash scripts can run perfectly without the `sh` extension.

2.2 Adding Shebang

Bash scripts start with a `shebang`. Shebang is a combination of `bash #` and `bang !` followed by the shell path. This is the first line of a script, this tells the shell to execute it via bash shell. Shebang is simply the absolute path to the bash interpreter. Below an example of shebang:

```
#!/usr/bin/bash
```

This is the most common path, but this can differ depending on your Linux distro. To find the path of your shell interpreter, run:

```
[username@host ~]$ which bash
```

2.3 First script

Start by creating a bash script file. In a UNIX system, use the `touch` command to create a new file:

```
touch first_script.sh
```

Open the file with your preferred editor and add the following commands:

```
1 #!/bin/bash
2 echo Today is $(date)
```

- Line 1: This is the `Shebang`
- Line 2: The `echo` command is displaying a string and a variable. We will come back to variables later.

To execute the script, run `./first_script.sh` or `sh first_script.sh` on your terminal. It might however not work, in that case, run this command:

```
[username@host ~]$ chmod +x first_script.sh
```

Here,

- `chmod` modifies the access permission and the special mode flags.
- `+x` adds the execution rights to the current user. This means that you can now run the script.

3 Basics

3.1 Comments

Comments start with a `#` in bash scripts.

```
# This is a comment
```

3.2 Conditional execution

It is possible to write multiple commands with the `&&` operator.

```
[username@host ~]$ git commit && git push   # Both commands executed
```

```
[username@host ~]$ git commit || echo "Commit failed"
```

3.3 Multiple commands

You can separate multiple commands by adding a `;` between them.

```
[username@host ~]$ pwd ; ls
/home/user/Documents
file1.txt file2.txt
```

4 Variables

Variable in bash, as in any other programming language, let you store data. In bash, they are no data types, variables can store numerical values, characters or string.

- Assign a value:

```
city=Stuttgart
age=26
welcome="Hello World!"
echo $city $age $welcome
```

```
Stuttgart 26 Hello World!
```

- Use an existing variable somewhere with the `$` sign

```
city="Stuttgart"
other_city=$city
echo $other_city
```

```
Stuttgart
```

⚠. Do not add spaces before and after the `=`.

4.1 Variable conventions

- Variables should start with a letter or an underscore
- Variable names can contain letters, number, and underscores
- Variable names are case-sensitive.
- Variable name should not contain spaces or special characters
- Avoid reserved keywords like `if`, or `else`

4.2 Numeric calculations

Here are a couple calculation examples:

```
$((a + 200))     # Add 200 to a
```

```
$(($RANDOM%200))  # Random number 0..199
```

```
declare -i count  # Declare as type integer
count+=1          # Increment
```

4.3 The `declare` key-word

The built-in declare statement does not need to be used to explicitly declare a variable in bash, the command is often employed for more advanced variable management tasks. `declare` syntax: `declare [options] [variable=name]="[value]"` Options:

-a	The variable is an indexed array. You cannot unset this attribute.
-A	The variable is an associative array. You cannot unset this attribute.
-f	Declare a bash function, not a variable.
-F	Display the function's name and attributes.
-g	Apply the global scope to all the variable operations inside a shell function. The option does not work outside shell functions.
-i	The value of the variable is an integer. Unset the attribute with +i.
-l	The variable name consists of lowercase characters only. Unset the attribute with +l.
-n	The variable becomes a name reference for another variable. Unset the attribute with +n.
-p	Display options and attributes of variables.
-r	The variable is read-only. Unset the attribute with +r.
-t	If used with functions, the item inherits DEBUG and RETURN traps from the parent shell. Unset the attribute with +t.
-u	The variable name consists of uppercase characters only. Unset the attribute with +u.
-x	Export the variable to child processes, similar to the export command. Unset the attribute with +x.

A few tips:

```
declare -i testvar="100" # We declare an integer of 100
declare -p | grep testvar
```

Output:

```
declare -- _="testvar=100"
declare -- testvar="100"
```

4.4 Arrays

Defining Arrays:

```
Fruits=('Apple' 'Banana' 'Orange')
Fruits[0]="Apple"
Fruits[1]="Banana"
Fruits[2]="Orange"
```

Working with arrays:

```
echo "${Fruits[0]}"           # Element #0
echo "${Fruits[-1]}"          # Last element
echo "${Fruits[@]}"           # All elements, space-separated
echo "${#Fruits[@]}"          # Number of elements
echo "${#Fruits}"             # String length of the 1st element
echo "${#Fruits[3]}"          # String length of the Nth element
echo "${Fruits[@]:3:2}"       # Range (from position 3, length 2)
echo "${!Fruits[@]}"          # Keys of all elements, space-separated
```

Operations:

```
Fruits=("${Fruits[@]}" "Watermelon")    # Push
Fruits+=('Watermelon')                  # Also Push
Fruits=( "${Fruits[@]/Ap*/}" )          # Remove by regex match
unset Fruits[2]                         # Remove one item
Fruits=("${Fruits[@]}")                 # Duplicate
Fruits=("${Fruits[@]}" "${Veggies[@]}") # Concatenate
lines=(`cat "logfile"`)                 # Read from file
```

Iterations:

```
for i in "${arrayName[@]}"; do
  echo "$i"
done
```

5 Special characters

5.1 Redirection

Before a command is executed, its input and output may be redirected using a special notation interpreted by the shell. Redirection allows commands' file handles to be duplicated, opened, closed, made to refer to different files, and can change the files the command reads from and writes to. Redirection may also be used to modify file handles in the current shell execution environment. The following redirection operators may precede or appear anywhere within a simple command or may follow a command. Redirections are processed in the order they appear, from left to right.

- The **Regular output `>` operator** is probably the most recognized of the operators. The standard output (stdout) is usually to the terminal window. It is usually used for Writing in a file:

```
[username@host ~]$ date
Tue Jun 11 13:10:03 CEST 2024
[username@host ~]$ date > spec.txt
[username@host ~]$ cat spec.txt
Tue Jun 11 13:10:03 CEST 2024
```

- The **Regular output append `>>` operator** adds the output to the existing content instead of overwriting it. This allows you to redirect the output from multiple commands to asingle file.

```
[username@host ~]$ date >> spec.txt
[username@host ~]$ pwd >> spec.txt
[username@host ~]$ cat spec.txt
Tue Jun 11 13:10:03 CEST 2024
/home/User/Documents/
```

- The **Regular input `<` operator** pulls data in a stream from a given source. This operator is especially useful for reading files (see later).

```
[username@host ~]$ wc -lc file.txt
2 52 file.txt
[username@host ~]$ wc -lc < file.txt
2 52
```

  Here only the content of the file is passed to the `wc` command.

- The **regular error `2>` operator** redirect standard errors. When a program or script does not generate the expected results, it throws an error. The error is usually sent to the **stdout**, but it can be redirected elsewhere. The **stderr** operator is `2>`

```
[username@host ~]$ png
bash: png: command not found...
[username@host ~]$ png 2> error.txt
[username@host ~]$ cat error.txt
bash: png: command not found...
```

- For redirecting the **Standard Output** and **Standard Error**, They are three formats: `&>`, `>&` and the `> FILE 2>&1`

```
[username@host ~]$ (pwd && png) > output.txt 2> err.txt
[username@host ~]$ (pwd && png) >& output.txt
[username@host ~]$ (pwd && png) &> output.txt
[username@host ~]$ cat output.txt
/home/user/Documents
[username@host ~]$ cat err.txt
-bash: -png: command not found
```

Here, the `(pwd && png)` create an error and a regular output. It is only here for example purposes.

From the two first forms, the `&>` is preferred.

- The **Here document** `<<` **operator** instruct the shell to read the input from the current source until a line containing only word (with no trailing blanks) is seen.

```
[username@host ~]$ wc << END
> One two three
> four
> five
> END
      3       5      24
```

Here, `END` can be replaced by anything.

- The **Here String** `<<<` is a variant of **Here Document**. The result is supplied as a single string, with a newline appended, to the command on its standard input. It is especially useful for passing command line argument to another as a string.

```
[username@host ~]$ ls
file1.txt file2.txt script1.sh script2.sh
[username@host ~]$ grep "txt" <<< $(ls)
file1.txt
file2.txt
```

Here, even though it isn't the regular way of doing it, the string of the `ls` command is passed to the `grep` command. More practical and useful examples will be cover in the `while` **loop** section.

- The **Pipe** `|` operator takes the output of the first command and makes it the input of the second command.

```
[username@host ~]$ ls
Document Download Pictures
[username@host ~]$ ls | grep "Do"
Documents
Downloads
```

The pipe operator is especially great for combining multiple commands.

5.2 Other special characters

- `"` & `'`: There is a difference between the single and double quote. Single quotes preserve literal meaning; double quotes allow substitutions. Examples:

```
a=apple
```

Bash	Output	Comment
`"$a"`	`apple`	variables are expanded inside `""`
`'$a'`	`$a`	variables are not expanded inside `''`
`"'$a'"`	`'apple'`	” has no special meaning inside `""`
`'"$a"'`	`"$a"`	”” is treated literally inside `''`
`'\''`	invalid	can not escape a `'` within `''`; use `"'"` or `$'\''` (ANSI-C quoting)
`'\"'`	`\"`	`\` has no special meaning inside `''`
`"redapple$"`	`redapple$`	`$` followed by no variable name evaluates to `$`
`"\'"`	`\'`	`\'` is interpreted inside `""` but has no significance for `'`
`"\""`	`"`	`\"` is interpreted inside `""`
`"*"`	`*`	glob does not work inside `""` or `''`
`'$arr[0]'`	`$arr[0]`	array access not possible inside `''`
`"$arr[0]"`	`apple`	array access works inside `""`

5.3 Glob

- `*`: **Glob**al. Match any single character (not between brackets).

```
[username@host ~]$ ls
Documents Downloads Pictures
[username@host ~]$ ls D*
Documents:
Doc

Downloads:
```

```
[username@host ~]$ ls
file1.txt script1.sh script.sh
[username@host ~]$ ls *.sh
script1.sh script.sh
```

- An expression ” `[...]` ” where the character after the leading `'['` is not an `'!'` matches a single character. By convention, two characters separated by `'-'` denote a range. (Thus, `[A-Fa-f0-9]` is equivalent to `[ABCDEFabcdef0123456789]`.)

6 Conditions

6.1 If and Else

We define a condition with the keywords `if`, `if-else` and/or `if-elif-else` for nested conditionals.

```
if [[ condition ]];
then
   statement
elif [[ condition ]]; then
    statement
else
    do this is default
fi
```

```
#!/bin/bash

echo "Please enter a number: "
read num

if [ $num -gt 0 ]; then
  echo "$num is positive"
elif [ $num -lt 0 ]; then
  echo "$num is negative"
else
  echo "$num is zero"
fi
```

A few rules:

- Always keep spaces between brackets and the comparison/check

  ```
  if [$foo -gt 3]; then
  ```

  This does **not work**.

- Always terminate the line before the `then` (Add a `;`)

- It is a good habit to quote string variables, otherwise they are likely to give trouble if they contain spaces and/or newlines.

  ```
  if [ "$foo" == "Foo" ]; then
  ```

- It is possible for **arithmetic expressions** only to use parenthesis.

  ```
  if (( $num <= 5 )); then
  ```

- The basic rule of bash when it comes to conditions is `0 equals true`, and `> 0 equals false`.

A few expressions for conditions. There are more than these below.

Primary	**Meaning**
EXPR1 `-a` EXPR2	AND
EXPR1 `-o` EXPR2	OR
`!` EXPR	Invert condition
EXPR1 `-gt` EXPR2	Greater than
EXPR1 `-lt` EXPR2	Less than
EXPR1 `-eq` EXPR2	Equal
EXPR1 `-le` EXPR2	Less or Equal
EXPR1 `-ge` EXPR2	Greater or Equal
`-a` FILE	True if FILE exists
`-b` FILE	True if FILE exists & is a block-special file
`-c` FILE	True if FILE exists & is directory
`-e` FILE	True if FILE exists
`-f` FILE	True if FILE exists & regular file
`-g` FILE	True if FILE exists & SGID bit is set
`-r` FILE	True if FILE exists & readable
`-z` STRING	True of the length if ”STRING” is zero.
`-n` STRING	True if the length of ”STRING” is non-zero.
STRING1 OP STRING2	Test between strings, with OP being `==`, `!=`, `>` or `<`

- It is easy to quickly test a condition:

  ```
  [ $foo -gt 3 ]; && echo true
  ```

  If true is printed, it means that your condition returned true.

Conditions with the double brackets `[[]]` serves as an enhanced version of the single-bracket syntax. It mainly has the same features, but also some important differences with it.

- The double brackets syntax features shell globing. This means that an asterisk `*` will expand to literally anything. For instance here, if you want to match 'foo' or 'Foo':

  ```
  if [[ "$foo" == *[fF]oo* ]]; then
  ```

- Word splitting is different, so that omitting quotes around string variables and use a condition makes no problems.

  ```
  if [[ $foo == Foo ]]; then
  ```

- Single and double quotes conditions handles differently the file name expansion (globbing) and testing.

  ```
  [ -a *.sh ]; then
  ```

  This above line will **not work** if multiple `.sh` files exists.

  - The single brackets `([ ... ])` are a synonym for the test command.
  - When using `[ -a ... ]`, the shell performs file name expansion on `*.sh` before passing it to the *test* command.
  - If there are multiple `*.sh` files in the directory, the expansion results in multiple arguments being passed.
  - As a result, it will fail if more than one `sh` file is present.

  However, `[[ -a *.sh ]]` will work

  - Whithin `[[ ... ]]`, file name expansion is not performed in the same way. Instead `[[ -a *.sh ]]` is evaluated as a single expression/
  - The pattern `[[ ... ]]` is more flexible and can handle the `*.sh` correctly

- It is possible to use more general expressions within `[[ ... ]]` like `==` or `&&`
- Double brackets allows **regex** pattern matching using the `=~` operator.

6.2 Case statement

The `case` statement is used to compare a given value against a list of pattern.

Syntax:

```
case expression in
    pattern1)
        # code
        ;;
    pattern2)
```

Example:

```
fruit="apple"
case $fruit in
    "apple")
        echo "This is a red fruit."
        ;;
```

```
        # code
        ;;
    *)
        # Default
        ;;
esac
```

```
    "banana")
        echo "This is a yellow fruit."
        ;;
    "orange")
        echo "This is an orange fruit."
        ;;
    *)
        echo "Unknown fruit."
        ;;
esac
```

7 Loop

7.1 For loop

The `for` loop has the following syntax:

First Syntax

```
for VAR in 1 2 3 .. N
do
    command1
    command2
    ..
    commandN
done
```

Second Syntax

```
for VAR in file1 file2
do
    command1 on $VARIABLE
    command2
    ..
    commandN
done
```

Third Syntax

```
for VAR in $(Linux Command here)
do
    command1 on $OUTPUT
    command2 on $OUTPUT
    ..
    commandN
done
```

Examples first syntax

```
#!/bin/bash
for i in 1 2 3 4 5
do
   echo "Welcome $i times"
done
```

```
Welcome 1 times
Welcome 2 times
Welcome 3 times
Welcome 4 times
Welcome 5 times
```

```
#!/bin/bash
for i in {1..5}
do
   echo "Welcome $i times"
done
```

```
Welcome 1 times
Welcome 2 times
Welcome 3 times
Welcome 4 times
Welcome 5 times
```

Since Bash 4.0+ it is allowed to use the following syntax.

```
#!/bin/bash
for i in {0..10..2}
do
   echo "Welcome $i times"
done

```

```
Welcome 0 times
Welcome 2 times
Welcome 4 times
Welcome 6 times
Welcome 8 times
Welcome 10 times
```

7.1.1 C shape for Loop

For loop can also share a common heritage with the C programming language.

```
for (( init; condition; step ))
do
    shell_commands
done
```

```
#!/bin/bash
for (( c=1; c<=5; c++ ))
do
   echo "Welcome $c times"
done
```

7.1.2 Notes

It is possible to use the keywords `break` to exit a loop, or `continue` for going to the next loop turn.

7.2 While loop

Syntax:

```
while CONDITION
do
    COMMANDS
done
```

They are several known use with the while command:

7.2.1 Read File

They couple strategies to read a file in bash. The most straightforward is to use the `cat` command: `content=$(cat file.txt)`
Since we cannot read line by line the content of the file, it is not the best way. To do so, we can use the while loop:

```
while read line; do
    Command
done < file.txt
```

Or:

```
while IFS= read -r line; do
    Command
done < file.txt
```

- `IFS` is the **I**nternal **F**ield **S**eparator. It prevent leading/trailing whitespace from being trimmed.
- As mentioned before, the `-r` option prevent backslash escapes from being interpreted

7.2.2 Read bash command line by line

In the same logic, it is possible to read line by line a bash command:

```
while read line; do
    echo "$line"
done <<<$(ps -au)
```

7.3 Until loop

Syntax:

```
until CONDITON
do
    COMMAND
done
```

7.4 Select loop

Allow to create a simple menu system. Format:

```
until ITEM in [LIST]; do
    COMMAND
done
```

It is possible to use the PS3 for prompting something at each loop turn right before a choice. Example:

```
#!/bin/bash
names='Kyle Cartman Stan Quit'
PS3='Select character: '
select name in $names; do
    if [ $name == 'Quit' ]
```

```
1) Kyle
2) Cartman
3) Stan
4) Quit
Select character: 1
```

```
    then
        break
    fi
    echo Hello $name
done
```

```
Hello Kyle
```

Here, we change the `PS3` variable to change the prompt that is displayed. By default, the `PS3=#?`

8 Functions

It is possible to declare functions in bash. They are two ways of defining functions:

Preferred and more used format:

```
function_name () {
    commands
}
```

Single line version:

```
function_name () { commands; }
```

Second version:

```
function function_name () {
    commands
}
```

Single line version:

```
function function_name () { commands; }
```

8.1 Return values

They are two ways of returning values:

With the return keyword:

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

The better option to return a value from a function is to send the value to `stdout` use `echo` of `printf` (See doc).

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

8.2 Arguments

To pass any number of arguments to the bash function simply put them right after the function's name, separated by a space. It is a good practice to double-quote the arguments to avoid the misparsing of an argument with spaces in it.

- The passed parameters are `$1`, `$2`, `$3` .. `$n`, corresponding to the position of the parameter after the function's name.
- The `$0` variable is reserved for the function's name.
- The `$#` variable holds the number of positional parameters/arguments passed to the function.
- The `$*` and `$@` variables hold all positional parameters/arguments passed to the function.
  - When double-quoted, `"$*"` expands to a single string separated by space (the first character of IFS) - `"$1 $2 $n"`.
  - When double-quoted, `"$@"` expands to separate strings - `"$1" "$2" "$n"`.
  - When not double-quoted, `$*` and `$@` are the same.

```
greeting () {
  echo "Hello $1"
}

greeting "Joe"
```