Lecture 8: Bash fancy features

Last update: 20200624

Table of Contents

1. Subshells: `( ... )`

We have already seen how the code block `{ ... }` can be used in **Bash** to embed the specific code snippet inside, and allocate only to the execution of that code snippet separate I/O facilities from the rest of the code. Another way of achieving this functionality is to use *subshell* `( ... )`. Any code block, when embedded within the round braces `( ... )`, forms the subshell. Generically, a subshell is defined as:

```
(
 command-input-1
 command-input-2
 ...
 command-input-n
)
```

or completely equivalently, with the one-liner:

```
( command-input-1; command-input-2; ... ; command-input-n; )
```

Just like it was done for the code block `{ ... }` in the previous sections, we now provide an executive summary of the most important features of subshell `( ... )`:

1. it inherits the current environment and cannot modify it globally
2. has its own `1>` and `2>` streaming facilities
3. starts a separate process, its PID can be obtained programmatically from the built-in variable **BASHPID**

From the above list, it is clear that there are a lot of similarities between `{ ... }` and `( ... )`. Basically, there are only two important differences:

1. when compared to the parent shell, `( ... )` starts a new process, while `{ ... }` does not

2. both `( ... )` and `{ ... }` inherit the environment from the parent shell, but `{ ... }` can modify it globally while `( ... )` cannot

When it comes to *stdout* and *stderr* streams, there is no difference between `( ... )` and `{ ... }`. The subshell `( ... )` is usually less efficient than the code block `{ ... }`, because it runs a separate process. However, since it cannot modify the environment in which it is run globally, `( ... )` is safer. As a rule of thumb, `( ... )` shall be preferred over `{ ... }` unless the efficiency is concern.

Typically, the subshells are executed in the background, by using the following generic syntax:

```
( command-input-1; command-input-2; ... ; command-input-n; ) &
```

The advantage of running subshells in the background is that now by using the **wait** command we can decide whether the rest of the code in the script will or will not wait the subshell execution to terminate (just like for any other command running in the background). This is very handy because we can use in that subshell automatically the already initialized environment in the script, execute the subshell, get back the result and keep environment unmodified.

The use case of subshell is illustrated with the following simple example:

```
( echo "Subshell PID: ${BASHPID}"; date; date -q; sleep 10m; ) 1>output.log
2>error.log &
```

Completely equivalently, the above code snippet could have been implemented across multiple lines:

```
(
 echo "Subshell PID: ${BASHPID}"
 date
 date -q
 sleep 10m
) 1>output.log 2>error.log &
```

It's a matter of personal taste which of the two versions is used in practice, but from the **Bash** perspective, they are the same. After executing, we see now with **jobs -l** the following printout:

```
[2]+  7941 Running                 ( echo "Subshell PID: ${BASHPID}"; date; date
-q; sleep 10m ) > output.log 2> error.log &
```

This means that the whole composite code inside the subshell now behaves like any other command running in the background. Any *stdout* printout in the body of subshell from any command is redirected with `1>` in the file 'output.log', whose content is:

```
Subshell PID: 7941
Do 4. Jul 07:37:55 CEST 2019
```

On the other hand, any *stderr* stream within subshell body was redirected with `2>` in the separate file 'error.log':

```
date: invalid option -- 'q'
Try 'date --help' for more information.
```

We can suspend the subshell execution just as we did it for individual commands:

```
kill -TSTP 7941
jobs -l
# [2]+  Stopped                 ( echo "Subshell PID: ${BASHPID}"; date; date -
q; sleep 10m ) > output.log 2> error.log
```

and resume it:

```
kill -CONT 7941
jobs -l
# [2]+  7941 Running                 ( echo "Subshell PID: ${BASHPID}"; date;
date -q; sleep 10m ) > output.log 2> error.log &
```

In the same fashion, we can terminate only the subshell execution, without affecting the script execution from which the subshell was launched. This is true whether or not subshell is executed in the background. For instance, if we have the following schematic situation:

```
some code
( some massive computation in subshell )
the remaining code waiting subshell to terminate
```

If the code execution got stuck because of some massive computation in the subshell, instead of terminating the whole script and restarting , we can terminate differentially only the subshell from a separate terminal, by sending some of the signals `TSTP`, `INT`, `QUIT` or `KILL` directly to the PID corresponding to the subshell. This is not possible for code block `{ ... }` because its PID is the same as the PID of the parent shell.

2. Process substitution operator: `<( ... )`

Process substitution operator `<( ... )` is a feature which is not provided by all shells, therefore its usage typically leads to some portability problems across different shells. Nevertheless, since it is supported by **Bash** and since it comes very handy in some frequently encountered cases in practice, we introduce it next.

Loosely speaking, process substitution operator translates on the fly the standard output stream of any command into a 'virtual' file, which then can be fed to the commands which accept only files as arguments. The true mechanism behind the scene is much more complicated than that (it relies on so-called 'named pipes'), but this loose explanation captures its behaviour in practice quite well.

Within `<( ... )` operator we can execute as many commands as we wish, separated with `;` delimiter. Because of this, the process substitution operator offers more flexibility than pipe `|`, which can forward directly the *stdout* of only one command as an input to another command.

In we execute multiple commands within `<( ... )` operator, the output of each command is summed up in one large common 'virtual' file, which then we can feed for further processing to commands like **grep**, **awk**, etc. This is best illustrated with the examples.

Example 1: How to extract only the text in the 4th column from the output of 3 consecutive commands? By using the process substitution operator and **awk**, the solution is very simple and elegant:

```
awk '{print $4}' <(command1; command2; command3)
```

In order to solve this problem without using the `<( ... )` operator, we would need to use something like:

```
command1 > someFile
command2 >> someFile
command3 >> someFile
awk '{print $4}' someFile
rm someFile
```

Clearly, by using the process substitution operator `<( ... )` we have saved quite some unnecessary coding steps.

Example 2: How to parse line-by-line through the output of some command and manipulate each line programmatically?

We have already seen that we can with the **while+read** construct parse line-by-line through the content of some external file. By using the process substitution operator `<( ... )`, we can trivially extend that functionality to the output of some command. For instance, let us do some programmatic manipulation on the output of **ls -al** command:

```
while read; do
 echo "In the line $REPLY there are ${#REPLY} characters"
done < <(ls -al)
```

The output of this command could look like:

```
In the line -rw-r--r--  1 abilandz abilandz  9508 Jul  2 09:42 bash_logo.png
there are 64 characters
In the line -rw-rw-r--  1 abilandz abilandz  7652 Jul  2 10:51 colours.png there
are 62 characters
```

This works because from the perspective of **while+read** construct, redirection from external file via `< someFile` or via process substitution operator `< <( ... )` is completely equivalent, because the output of `<( ... )` is essentially a 'virtual file'.

Example 3: How to check programmatically if the printouts of two commands are exactly the same?

In order to compare the content of two files, we can use the **diff** command, with the following generic syntax:

```
diff file1 file2 && echo same || echo different
```

With the process substitution operator, we can extend the functionality of **diff** command to the comparison between the printout of different commands. The generic use case could look like:

```
diff <(command1) <(command2) && echo same || echo different
```

Without the process substitution operator, we would need to dump the printout of each command in some temporary file, and then compare the content of those temporary files with **diff** command. Clearly, process substitution operator saves also in this frequently encountered example a lot of additional and completely trivial coding.

3. Here strings: `<<<`

'Here strings' is a **Bash** feature which can save a bit of typesetting in some frequently encountered use cases. For instance, with 'here strings' we can feed the already made string directly into the command, without typing that string manually. Some commands (e.g. **bc**), we first need to start, and only then supply the string as an input interactively from the keyboard. Such cases can also be bypassed with 'here strings' and instead of typing directly from the keyboard, the commands can process the input string programmatically.

The generic syntax for the usage of 'here strings' can be represented in the following schematic way:

```
command <<< "some hardwired string"
```

In the above example, the input string is hardwired, but it can be also supplied as the content of variable:

```
Var="some hardwired string"
command <<< ${Var}
```

The command substitution operator `$( ... )` and arithmetic expansion `$(( ... ))` can be used as well on the right hand side of `<<<` operator. What **Bash** is doing when it encountered `<<<` can be summarized as follows: Whatever is on the right hand side of `<<<` undergoes expansion (e.g. `${Var}` is replaced with variable content, `$(( ... ))` is replaced with the result of arithmetic evaluation, etc.), and then the resulting expression is simply fed to the 'stdin' of **command**. The result of expansion is supplied as a single string to **command**, with a newline always appended.

It is also possible to think about 'here strings' as being the shortcut notation for the following common construct:

```
echo "someString" | command
```

because the output of above line is the same as of:

```
command <<< "someString"
```

Therefore, instead of using the command **echo** in combination with pipe `|`, we can simply condense the syntax and use only `<<<`, the final printout is exactly the same.
Example 1: Let's define `Var=20180524`. How to check programmatically with **grep** if this string begins with the pattern '2018'?

The two solutions below yield the same result:

```
echo ${Var} | grep "^2018"
grep "^2018" <<< ${Var}
```

'Here strings' are frequently combined with **sed**, that is illustrated in the next example.

Example 2: Let's define `Var=20180524`. How to change pattern '2018' into '2020' programmatically and immediately redefine the variable `Var` to the new content?

```
sed "s/2018/2020/" <<< $Var
echo ${Var}
# prints 20180524, content of 'Var' is unchanged
```

Note that the content of the starting variable `Var` is still '20180524', we only saw the pattern replacement in the printout on the screen. If we want to update immediately the content of the starting variable, this can be achieved with:

```
Var=$(sed "s/2018/2020/" <<< $Var)
echo ${Var}
# prints 20200524, content of 'Var' is changed
```

Finally, we illustrate the typical usage of 'here strings' in combination with **bc** command, to perform floating point arithmetic's. We start by recalling the example from Lecture 6.

Example 3 (revised from Lecture 6): How can we divide 10/7 at the precision of 30 significant digits? Solution was given previously by the following code snippet:

```
echo "scale=30; 10/7" | bc
```

However, this problem can be also solved a bit simpler with the usage of 'here strings':

```
bc <<< "scale=30; 10/7"
```

The output in both cases is the same and it reads:

```
1.428571428571428571428571428571
```

There are the cases in which **echo** + `|` is more efficient, while there are also the cases in which the 'here strings' `<<<` run faster, so both versions are used frequently in practice.

4. Here documents: `<<`

'Here documents' are typically used within shell scripts to write programmatically entire files, including the new scripts. Programmatically written files with 'here documents' can be used as a further input to the script which made them or as an input to some other commands. Programmatically written scripts with 'here documents' can be immediately executed in the script which created them.

There are three important use cases of 'here documents'. The first one uses the following generic syntax:

```
cat > someFile <<HERE-DOC
... some code here ...
HERE-DOC
```

The 'here document' begins with `<<` and its body is marked with the matching pair of delimiters. The name of the delimiters is arbitrary, as long as the closing one does not coincide with some string in the code in the body. The above construct evaluates all code inside delimiters named 'HERE-DOC', and dumps it in the external file named 'someFile'. There is nothing special about the choice of 'HERE-DOC' to name the delimiters, we could have uses as well something like

```
cat > someFile <<EOF
... some code here ...
EOF
```

The empty characters at the end of delimiters matter, and this is a typical source of errors. If the name of the opening delimiter is 'HERE-DOC' (no empty characters!) and the name of the closing delimiter is 'HERE_DOC ' (one empty character at the end!), we will get an error, as the two delimiters do not match exactly each other. Therefore, make sure there are no trailing empty characters after the delimiters.

With the above version of 'here documents', **Bash** supports variable and command substitution in the body of 'here documents', which enables to write files programmatically. This is best illustrated with the following simple script named 'testHD.sh', which takes as two arguments two floats, sums them up, and dumps everything (alongside with some other info) in the external file named 'output.log':

```
#!/bin/bash

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

cat > output.log <<HERE-DOC
Today is: $(date)
File produced by script: ${BASH_SOURCE}
Sum is: $(bc <<< "scale=2; $1 + $2")
HERE-DOC

return 0
```

If we execute this script for instance with:

```
source testHD.sh 2.44 10.23
```

this script has by using the 'here document' created an external file named 'output.log', whose content is:

```
Today is: Tue Jun 16 14:04:52 CEST 2020
File produced by script: testHD.sh
Sum is: 12.67
```

Note that we did not need to use **echo** in the body of 'here documents' to dump any text into the file 'output.log'. On the other hand, if we would have used code block `{ ... }` to achieve the same results, the code inside the code block would be clogged with **echo** statements:

```
#!/bin/bash

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

{
 echo Today is: $(date)
 echo File produced by script: ${BASH_SOURCE}
 echo Sum is: $(bc <<< "scale=2; $1 + $2")
} > output.log

return 0
```

Because of this difference, 'here documents' are simpler and more elegant when writing programmatically the files than the code blocks `{ ... }`.

The second important use of 'here documents' uses the following generic syntax:

```
cat > someFile.log <<'HERE-DOC'
... some code here ...
HERE-DOC
```

Note the use of quotes round the opening delimiter. In this version, the code in the body of 'here documents' is literally evaluated with the following phrase: What you typed is what you get. The meaning of all special symbols in the code in the body of 'here document' is killed if the starting delimiter is enclosed within quotes. For instance, if we modify the previous example:

```
#!/bin/bash

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

cat > newScript.sh <<'HERE-DOC'
Today is: $(date)
File produced by script: ${BASH_SOURCE}
Sum is: $(bc <<< "scale=2; $1 + $2")
HERE-DOC

return 0
```

and execute it as before, the content of the file 'newScript.sh', to which we have redirected now the content of 'here document', is dramatically different:

```
Today is: $(date)
File produced by script: ${BASH_SOURCE}
Sum is: $(bc <<< "scale=2; $1 + $2")
```

This is perfectly valid **Bash** source code. Literally, what we type in the body of this version of 'here document' is what we get in the external file.

The above version of 'here documents' has an obvious and important use case: We can within one **Bash** script programmatically write (and execute immediately if necessary) another **Bash** script, with all special characters and **Bash** syntax in place. This feature is more general, because with this version of 'here documents' we can preserve the special syntax of any other programming language, dump the code in external file, compile it and use executable immediately in the very same **Bash** script in which you have written the initial source code.

Finally, 'here documents' can be used to comment out multiple lines of **Bash** source code in one go, instead of using `#` at the beginning of each line. This is illustrated with the following example:

```
... code line 1 ...
... code line 2 ...
... code line 3 ...
... code line 4 ...
... code line 5 ...
... code line 6 ...
... code line 7 ...
... code line 8 ...
... code line 9 ...
```

How for instance we would comment out the code in the lines 2 to 8, and execute only the code in the lines 1 and 9? This can be achieved in the following way:

```
... code line 1 ...
: <<'HERE-DOC'
... code line 2 ...
... code line 3 ...
... code line 4 ...
... code line 5 ...
... code line 6 ...
... code line 7 ...
... code line 8 ...
HERE-DOC
... code line 9 ...
```

We have essentially declared lines from 2 to 8 to be the body of 'here document', and then its content redirected to the infamous 'do-nothing' command `:`. This is yet another very neat use case of 'do-nothing' command! When using 'here documents' to comment out piece of the code, the version in which the opening delimiter is enclosed in quotes is safer, as it will prevent any potential variable or command substitution in the body, before passing the code over to 'do-nothing' command.

As this is the frequent source of painful debugging, we close this section by remarking again that the closing delimiter in 'here documents' shall not be followed by any trailing empty character.

5. Menus

Menus are built in **Bash** programmatically with the built-in command **select**, with the following generic syntax:

```
select someVariable in someList; do
 ... some commands ...
 break
done
```

The menu for selection is specified via the list, schematically indicated as 'someList', which is placed between the key word **in** and semicolon `;`. The list can be specified by hardwiring some entries separated with empty characters, by using the command substitution operator `$( ... )`, brace expansion mechanism, etc. If the the key word **break** is omitted, menu is offered again

and again for selection.

We illustrate now with a few concrete examples how to build menus in **Bash**. We start by saving the following code in the file 'selectCountry.sh'

```
#!/bin/bash

select Country in Germany Italy France Spain England; do
 echo "You have selected: $Country"
 break
done

return 0
```

After executing the script via:

```
source selectCountry.sh
```

the following menu appears on the screen:

```
1) Germany
2) Italy
3) France
4) Spain
5) England
#?
```

Bash has offered us with the predefined menu, and is now waiting for reply. If we for instance type `1` and press `Enter` we get as a result the following printout:

```
You have selected: Germany
```

Since we have used the key word **break**, the **select** command bails out immediately after the first selection in the menu was done.

As another example, we can build menu from the output of some command:

```
#!/bin/bash

select File in $(ls *.sh); do
 echo "You have selected script: $File"
 source $File
 break
done

return 0
```

The above code offers on the menu all scripts in the current working directory, and then we can directly execute the script from menu, literally with one key stroke, instead of typing **source scriptName**.

The default prompt symbol in **select** is `#?`, while the default prompt symbol in **Bash** is `$`. How the prompt will look like is determined from the content of special environment variables **PS1** (for **Bash**), and **PS3** (for **select**). For instance, we can re-execute the previous example with:

```
PS3="What is your choice? " source selectCountry.sh
```

The menu is now:

```
1) Germany
2) Italy
3) France
4) Spain
5) England
What is your choice?
```

As a concluding remark, we indicate that if the key word **in** is omitted and the list is not specified, the list is defaulted to the positional parameters supplied to the script or function, and menu is built out of all supplied positional parameters.