Introduction to Bash scripting language

Day 2

Alessandro Sciarra

Z02 – Software Development Center

08.10.2019

GOETHE UNIVERSITÄT FRANKFURT AM MAIN

CRC-TR 211
Strong-interaction matter under extreme conditions

Topics of the day

1. Loops
2. Choices
3. Colours and cursor movements
4. References and indirect expansion
5. Arrays and associative arrays
6. Input and output
7. File descriptors and redirection
8. Ready to script!?

You already know the basics

Although only one third of the course is over, you learnt already almost everything about the core of Bash. Today we will complete ~90% of the picture.

Loops

Ice cave in the Breiðamerkurjökull

Conditial loops

Often we need to repeat things: Copy and paste is not the solution!

`while` Repeat as long as a command is executed successfully (exit code is 0)

`until` Repeat as long as a command is executed unsuccessfully (exit code is not 0)

`for` It comes into two versions

- to iterate over a list
- to iterate over an integer index as in C

Which one should be used?

There will nearly always be multiple approaches to solving a problem. The test of your skill soon won't be about solving a problem as much as about how best to solve it.

While instead of until

The `until` loop is barely ever used, if only because it is pretty much exactly the same as `while !`

The `while` and `until` keywords

```
while COMMAND; do
    # Body of the loop: entered if COMMAND's exit code = 0
done

until COMMAND; do
    # Body of the loop: entered if COMMAND's exit code ≠0
done
```

- Testing command like `[` or preferably `[[` are often used
- Infinite loops can be achieved using the builtins `true`, `false` and `:`
- Use the `continue` builtin to skip ahead to the next iteration of a loop without executing the rest of the body
- use the `break` builtin to jump out of the loop and continue with the script after it
- Both `continue` and `break` accept an optional integer to act on nested loops

The while and until keywords

```
$ while true; do    # while :; do
>   echo "Infinite loop"
>   sleep 3
> done
Infinite loop
Infinite loop
^C    # Press CTRL-C
```

```
# An example of countdown...
$ deadline=$(date -d "8 seconds" +'%s'); \
> now=$(date +'%s'); \
> while (( deadline - now > 0 )); do
>   echo "$((deadline - now)) seconds to BOOM!"
>   sleep 3
>   now=$(date +'%s')
> done; echo 'BOOOOOM!!'
8 seconds to BOOM!
5 seconds to BOOM!
2 seconds to BOOM!
BOOOOOM!!
```

The `for` keywords

```
for VARIABLE in WORDS; do
    # Body of the loop: VARIABLE set to WORD
done

for (( EXPR1; EXPR2; EXPR3 )) # Expressions can be empty
    # Body of the loop
done
```

- In the second form,
  1. it starts by evaluating the first arithmetic expression;
  2. it repeats as long as the second arithmetic expression is successful;
  3. at the end of each loop evaluates the third arithmetic expression.
- Bash takes the characters between `in` and the end of the line and
  - it splits them up into words;
  - this splitting is done on spaces and tabs, just like argument splitting;
  - if there are any unquoted substitutions, they will be word-split as well (using `IFS`).

The for keywords

```
$ for (( ; 1; )); do echo "Infinite loop"; sleep 1; done
Infinite loop
Infinite loop
^C    # Press CTRL-C
```

```
$ for index in {0,1}{0,1}; do
>   echo "${index} in base 2 is $(( 2#${index})) in base 10"
> done; unset index
00 in base 2 is 0 in base 10
01 in base 2 is 1 in base 10
10 in base 2 is 2 in base 10
11 in base 2 is 3 in base 10
```

```
# BAD code!

$ for file in $(ls *.mp3); do    # AAAARGH!
>   rm "$file"
> done; unset file
```

The `for` keywords

```
$ ls
Happy birthday.mp3
$ for file in $(ls *.mp3); do    # AAAARGH!
>   rm "$file"
> done; unset file
rm: cannot remove `Happy': No such file or directory
rm: cannot remove `birthday.mp3': No such file or directory
```

You want to quote it, you say?

```
$ ls
Happy birthday.mp3   Hello.mp3
$ for file in "$(ls *.mp3)"; do    # AAAARGH!
>   rm "$file"
> done; unset file
rm: cannot remove `Happy birthday.mp3   Hello.mp3': No such
    file or directory
```

So, what do we do?

The `for` keywords

Use globs!

Bash **does** know that it is dealing with filenames, and it **does** know what the filenames are, and as such it can split them up nicely!

```
$ ls
Happy birthday.mp3   Hello.mp3
$ for file in *.mp3; do    # GOOD code
>   rm "$file"
> done; unset file
```

Do not be tempted

You might argue that, if there are no spaces in filenames, then you would have no troubles. But would you throw in the air a sharp knife trying to catch it afterwards, just because if you do not touch the blade it would be safe?

Choices

The Skaftafell natural park: Svartifoss

The `case` keyword

Completely equivalent to a `if-elif-else-fi` construct, but often handy:

```
case WORD in
    [ [(] pattern [| pattern]...) command-list ;;]...
esac
```

- The command-list corresponding to the first pattern that matches word is executed
- The match is performed according Pattern Matching rules
- The `|` is used to separate multiple patterns
- The `)` operator terminates a pattern list
- A list of patterns and an associated command-list is known as a **clause**

Before matching is attempted:

WORD under-goes

- tilde expansion
- parameter expansion
- command substitution
- arithmetic expansion
- quote removal

Pattern under-goes

Using `;&` or `;;&` instead of `;;` allows to execute the command list or to test the patterns of the next clause, respectively.

The `case` keyword

Completely equivalent to a `if-elif-else-fi` construct, but often handy:

```
case WORD in
    [ [(] pattern [| pattern]...) command-list ;;]...
esac
```

```
case ${LANG} in
    en* )
        echo 'Hello!' ;;
    fr* )
        echo 'Salut!' ;;
    de* )
        echo 'Hallo!' ;;
    it* )
        echo 'Ciao!' ;;
    es* )
        echo 'Hola!' ;;
    C | POSIX )
        echo 'Hello World!' ;;
    *)
        echo 'I do not speak your language.' ;;
esac
```


Using ;& or ;;& instead of ;; allows to execute the command list or to test the patterns of the next clause, respectively.

The `select` keyword

A convenient statement for generating a menu of choices that the user can choose from:

```
select NAME in WORDS; do
    # Body with commands -> break needed to terminate
done
```

- The list of words following in is expanded, generating a list of items
- The set of expanded words is printed on the standard error, each preceded by a number
- If the `in WORDS` is omitted, the positional parameters are printed, as if `in "$@"`
- The `PS3` prompt is then displayed and a line is read from the standard input:
  - If the line consists of a number corresponding to one of the displayed words, then the value of `NAME` is set to that word
  - If the line is empty, the words and prompt are displayed again
  - Any other value read causes `NAME` to be set to null
  - The line read is saved in the variable `REPLY`
- It is good practice not to `break` if the user typed something meaningless!

The `select` keyword

A convenient statement for generating a menu of choices that the user can choose from:

```
select NAME in WORDS; do
    # Body with commands -> break needed to terminate
done
```

```
$ select file in *; do
>     echo "You picked \"${file}\" (${REPLY})"
>     break
> done
1) Day_1.tex
2) Day_2.tex
3) Day_3.tex
#? 2
You picked "Day_2.tex" (3)
$ echo "After select: \"${file}\" (${REPLY})"
After select: "Day_2.tex" (3)
$ unset file
```

The `select` keyword

A convenient statement for generating a menu of choices that the user can choose from:

```
select NAME in WORDS; do
    # Body with commands -> break needed to terminate
done
```

```
$ select file in *; do
>     echo "You picked \"${file}\" (${REPLY})"
>     break;
> done
1) Day_1.tex
2) Day_2.tex
3) Day_3.tex
#? dlfagfdsu
You picked "" (dlfagfdsu)
$ echo "After select: \"${file}\" (${REPLY})"
After select: "" (dlfagfdsu)
$ unset file
```

It is always important to **validate the answer**!

The `select` keyword

A convenient statement for generating a menu of choices that the user can choose from:

```
select NAME in WORDS; do
    # Body with commands -> break needed to terminate
done
```

```
$ select file in *; do
>     if [[ ${file} != '' ]]; then  # GOOD PRACTICE
>         echo "You picked \"${file}\" (${REPLY})"
>         break;
>     fi
> done; unset file
1) Day_1.tex
2) Day_2.tex
3) Day_3.tex
#? dlfagfdsu
#? dgasdf
#? 4
#? 231345134
#? 2
You picked "Day_2.tex" (2)
```

Colours and cursor movements

Icebergs on the beach of the Jökursárlón

Colours and formatting in the terminal

🔗 A good reference

- Most terminals are able to display colours and formatted texts thanks to escape sequences
- Your script can benefit from using colours (e.g. errors, warnings, info)
- You might run into compatibility problems (not really) 🔗 Compatibility list
- The `echo` command has a `-e` option to enable the parsing of the escape sequences
- The `printf` command parses escape sequences

Escape sequences

`<Esc>[FormatCodem`

- The `<Esc>` character can be obtained with any of the following syntaxes:
  - `\e`
  - `\033`
  - `\x1B`
- The `FormatCode` is an integer and different ones can be combined using semi-colons

The colors can vary depending of the terminal configuration!

Colours and format codes

`<Esc>[FormatCodem`

0	Reset, all attributes off
1	Bold (or increased intensity)
2	Faint (decreased intensity)
3	Italic { not widely supported, sometimes treated as inverse }
4	Underline
5 or 6	Slow/Rapid Blink
7	Swap foreground and background colours
8	Hidden (useful for passwords)
2{1..8}	Reset second digit format (e.g. 24 to stop underlining)*
30 to 37	Set foreground colour } 8 colours
40 to 47	Set background colour
38;5;{0..255}	Set foreground colour } 256 colours
38;5;{0..255}	Set background colour
90 to 97	Set bright foreground colour } 16 colours
100 to 107	Set bright background colour

* GNOME Terminal 3.28 (VTE 0.52), debuting in Ubuntu 18.04 LTS, adds support for a few more styles. Use code 22 to unbold since 21 is double underline!

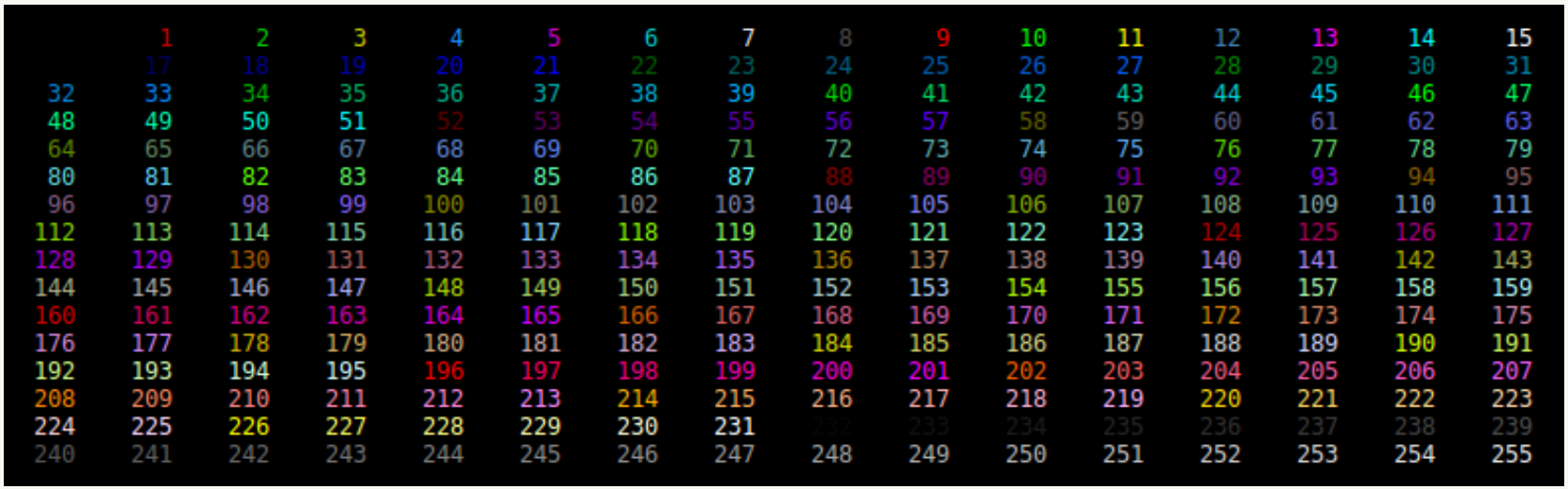
Colours and format codes

```
$ echo "Default \e[31mRed\e[0m"
Default \e[31mRed\e[0m
$ echo -e "Default \e[31mRed\e[0m"
Default Red
$ echo -e "Default \e[91mLight Red\e[0m"
Default Light Red
$ echo -e "Default \e[46mCyan\e[0m"
Default Cyan
$ echo -e "Default \e[106mLight Cyan\e[0m"
Default Light Cyan
$ echo -e "Default \e[1mBold\e[0m"
Default Bold
$ echo -e "Default \e[4mUnderlined\e[0m"
Default Underlined
$ echo -e "Default \e[4\e[91mUnderlined\e[0m"
Default Underlined
$ echo -e "Default \e[4;91mUnderlined\e[0m"
Default Underlined
$ printf "Default \e[4;91mUnderlined\e[24m still red\e[0m Default"
Default Underlined still red Default
$ echo -e "\e[40;38;5;82m Hello \e[30;48;5;82m World \e[0m"
 Hello  World
```


* GNOME Terminal 3.28 (VTE 0.52), debuting in Ubuntu 18.04 LTS, adds support for a few more styles. Use code 22 to unbold since 21 is double underline!

Colours and format codes

```
$ for fgbg in 38; do  # 38 48 to get also background
>     for color in {0..255}; do
>         printf "\e[${fgbg};5;%sm  %3s  \e[0m" ${color} ${color}
>         if [[ $(((color + 1) % 16)) -eq 0 ]]; then
>             echo
>         fi
>     done
>     echo
> done
```



* GNOME Terminal 3.28 (VTE 0.52), debuting in Ubuntu 18.04 LTS, adds support for a few more styles. Use code 22 to unbold since 21 is double underline!

Defining variables for colour codes

```
# Reset
Default='\033[0m'
# Regular Colors
Black='\033[0;30m'
Red='\033[0;31m'
Green='\033[0;32m'
Yellow='\033[0;33m'
Blue='\033[0;34m'
Magenta='\033[0;35m'
Cyan='\033[0;36m'
White='\033[0;37m'                    # Or you can wait to
# Bold                                # learn about functions
Bold='\033[1m'                        # and create your way!
BBlack='\033[1;30m'
BRed='\033[1;31m'
BGreen='\033[1;32m'
BYellow='\033[1;33m'
BBlue='\033[1;34m'
BMagenta='\033[1;35m'
BCyan='\033[1;36m'
BWhite='\033[1;37m'                 # ...and so on and so forth!
```

Cursor movements

In the same spirit of colours, the terminal cursor position can be moved:

`<Esc>[<L>;<C>H`	Puts the cursor at line `L` and column `C`
`<Esc>[<L>;<C>f`	Puts the cursor at line `L` and column `C`
`<Esc>[<N>A`	Move the cursor up N lines
`<Esc>[<N>B`	Move the cursor down N lines
`<Esc>[<N>C`	Move the cursor forward N columns
`<Esc>[<N>D`	Move the cursor backward N columns
`<Esc>[2J`	Clear the screen, move to (0,0)
`<Esc>[K`	Erase to end of line
`<Esc>[s`	Save cursor position
`<Esc>[u`	Restore cursor position

The `tput` command is a kind of wrapper for everything we learn in this section → Starting point

Cursor movements

Use your imagination to take advantage of this functionality:

```
$ while true; do
>     printf "     $(date) \n\e[1A"
>     sleep 1
> done
^C   Thu 11 Jul 18:07:26 CEST 2019 # Stop it with CTRL-C

# Here a crazy progress bar:
$ while true; do
>     printf '\e[s'
>     printf '%0.s=' $(seq 1 $(bc -l <<< "$RANDOM/32767*100"))
>     sleep 0.2
>     printf '\e[u\e[K'  # Use \r instead of saving cursor
> done
======================^C # Stop it with CTRL-C

$ printf 'You will never see my ${password}... MUAHAHA!\r\e[K'
$
```


The `tput` command is a kind of wrapper for everything we learn in this section → Starting point

References and indirect expansion

The geyser Strokkur: 25-35m high every ~10 minutes

The nameref attribute

Introduced in 2014, Bash v4.3.0

Reference: `declare -n reference`
The variable is a reference to another variable

- It allows variables to be manipulated indirectly
- Whenever the `reference` variable
  - is referenced, assigned to, unset,
  - or has its attributes modified (other than using or changing the **nameref** attribute itself),

  the operation is performed on the variable specified by the `reference`'s value!

```
$ aVar='Hello'; echo "${aVar}"
Hello
$ declare -n bVar='aVar'; echo "${bVar}"
Hello
$ bVar='Goodbye'; echo "${aVar}"
Goodbye
$ declare +n bVar; echo "${bVar}"
aVar
$ unset aVar bVar
```

The **nameref** attribute

Introduced in 2014, Bash v4.3.0

Think before using indirection

Putting variable names or any other bash syntax inside parameters is frequently done incorrectly and in inappropriate situations to solve problems that have better solutions. **It violates the separation between code and data**, and as such puts you on a slippery slope toward bugs and security issues. Indirection can make **your code less transparent and harder to follow**.

Normally, in bash scripting, you won't need indirect references at all. Generally, people look at this for a solution when they don't understand or know about Bash Arrays or haven't fully considered other Bash features such as functions.

Greg's Wiki

Sometimes you might need it

A **nameref** is commonly used within shell functions to refer to a variable whose name is passed as an argument to the function.

Indirect expansion

An alternative to use the **nameref** attribute

- It is about using the content of a variable as name of a different variable
- It is a particular case of parameter expansion: `${!parameter}`
  - Bash uses the value formed by expanding `parameter` as the new parameter
  - this is then expanded and that value is used in the rest of the expansion

```
1 $ aVar='Hello'; bVar='aVar'
2 $ echo "bVar contains \"${bVar}\" which contains \"${!bVar}\""
3 bVar contains "aVar" which contains "Hello"
4 $ unset aVar bVar
```

Indirect expansion

An alternative to use the **nameref** attribute

- It is about using the content of a variable as name of a different variable
- It is a particular case of parameter expansion: `${!parameter}`
  - Bash uses the value formed by expanding `parameter` as the new parameter
  - this is then expanded and that value is used in the rest of the expansion

```
1 $ aVar='Hello'; bVar='aVar'
2 $ echo "bVar contains \"${bVar}\" which contains \"${!bVar}\""
3 bVar contains "aVar" which contains "Hello"
4 $ unset aVar bVar
```

- If a `*` or a `@` is put at the end of parameter, the behaviour changes!

`${!prefix@}`
`"${!prefix@}"`
`${!prefix*}`
} Expands to the names of variables whose names begin with `prefix` as single word

`"${!prefix*}"` ⟶ As above, but words are separated by the first character of the `IFS`

Indirect expansion

An alternative to use the **nameref** attribute

- It is about using the content of a variable as name of a different variable
- It is a particular case of parameter expansion: `${!parameter}`
  - Bash uses the value formed by expanding `parameter` as the new parameter
  - this is then expanded and that value is used in the rest of the expansion

```
$ aVar='Hello'; bVar='aVar'
$ echo "bVar contains \"${bVar}\" which contains \"${!bVar}\""
bVar contains "aVar" which contains "Hello"
$ unset aVar bVar
```

- If a `*` or a `@` is put at the end of parameter, the behaviour changes!

`${!prefix@}`
`"${!prefix@}"`
`${!prefix*}`
} Expands to the names of variables whose names begin with `prefix` as single word

`"${!prefix*}"` ⟶ As above, but words are separated by the first character of the `IFS`

You will hardly need this!

Arrays and associative arrays

A baby moulin on the Breiðamerkurjökull

Motivation

- **Strings** are without any doubt the most used parameter type
- They are also the most **misused** parameter type, though!
- Strings hold **just one** element!
- Capturing a list in a string is very often plain wrong, even if it might work...
- A parameter contains just one string of characters, no matter the meaning of them
- If you put multiple filenames in a string, maybe to iterate over them at a later point, which delimiter would you use?
- Is there a delimiter that is not accepted to be part of a filename? Mmmh...

```
# This does NOT work in the general case
$ files=$(ls ~/*.jpg); cp ${files} /backups/    # BAD code!
```

Array

An array is a numbered list of strings: It maps integers to strings.

For filenames in Linux, there are usually almost no restrictions, apart from / and \0, you are allowed to use anything.

Creating an array

`array=(...)` The most used syntax: space separated list in parenthesis

```
array=(one two "three and four") # Equivalent to:
array=([0]=one [1]=two [2]="three and four")
# Sparse array perfectly legal!
array=([0]=First [11]=Middle [23]=Last)
```

`array[n]=` Setting single elements works for undeclared variable, too

```
array[0]=First
array[11]=Middle
array[23]=Last
```

`declare -a array` Rarely used but possible

```
declare -a array  # Because of declare,
array=One         # equivalent to array[0]=One, AVOID!
array[1]=Two
```

`array+=(...)` Concatenate r.h.s. array to l.h.s. array

The command `read -a array` allows to store what has been read in an array.

Storing filenames into an array

Take home message

If you want to fill an array with filenames, then you'll probably want to use **globs** in there!

```
$ files=(~/"My Photos"/*.jpg)
```

- Here we quoted the `"My Photos"` part because it contains a space
- If we hadn't quoted it, Bash would have split it up into

  ```
  files=('~/My' 'Photos/'*.jpg)
  ```

  which is obviously not what we want!
- We quoted **only** the part that contained the space: We cannot quote the `~` or the `*`
- If we do, they'll become literal and Bash won't treat them as special characters anymore!

Storing filenames into an array

Take home message

If you want to fill an array with filenames, then you'll probably want to use **globs** in there!

```
# Please, really, use globs instead of commands!

$ files=$(ls)       # BAD, BAD, BAD!
$ files=($(ls))     # STILL BAD!

# So, how should you do?

$ files=(*)         # GOOD!
```

Globs know about files

Using glob patterns is possible to create an array with filenames in different entries!

Accessing array's content (I)

Definition, single elements and length

`declare -p array` It prints the definition of the variable { not so useful in scripts }

```
$ declare -p array
declare -a array='([0]="one" [1]="two and four")'
```

`${array[n]}` It retrieves the n-th element { no error if entry missing in array }

```
$ echo "${array[1]}"; echo "_${array[3156]}_"
two and four

--
$ echo "${array[0+1]}" # [...] is an arithmetic context
two and four
```

`${#array[@]}` It retrieves the length of the array { same as `${#array[*]}` }

```
$ echo "${#array[@]}"; array[7]=Hello; echo "${#array[@]}";
2  # Quoting the expansion does not change anything!
3
$ echo "${#array[7]}"; echo "${#array[-1]}";
5
5  # What does this mean?
```


Curly braces in parameter expansions accessing arrays are mandatory.

Accessing array's content (II)

All elements at once

`"${array[@]}"` Bash replaces this syntax with each element properly quoted

```
$ printf '%s\n' "${array[@]}"
one
two and four
Hello
```

`"${array[*]}"` expands to a `IFS`-first-character-separated list of the array entries

```
$ IFS=':'; printf '%s\n' "${array[*]}"; unset IFS
one:two and four:Hello
```

Unquoted `${array[@]}` and `${array[*]}` let word splitting kick in!

```
$ printf '%s\n' ${array[@]} # The same with ${array[*]}
one
two
and
four
Hello
```

Accessing array's content (III)

All indices

`"${!array[@]}"` Bash replaces this syntax with the indices of the entries that are set

```
$ printf '%d\n' "${!array[@]}"
0
1
7
```

`"${!array[*]}"` expands to a `IFS`-first-character-separated list of the array indices

```
$ IFS=':'; printf '%s\n' "${!array[*]}"; unset IFS
0:1:7
```

In any case, prefer quoted versions!

Unquoted `${!array[@]}` and `${!array[*]}` let word splitting kick in.
This, for **normal** arrays, is harmless, since indices are just integers.

Iterating over array elements

- Depending on the needs you can iterate over the elements or the indices in the array
- When iterating over the **elements**, 99% of the times you will need `"${array[@]}"`
- When iterating over the **indices**, 99% of the times you will need `"${!array[@]}"`
- To implicitly iterate over arrays is possible using different commands (e.g. `printf`, `cp`)

```
$ array=(one "two and four")
$ for entry in "${array[@]}"; do
>     echo "${entry}"
> done
one
two and four
$ for index in "${!array[@]}"; do
>     echo "array[${index}]=${array[index]}"
> done
array[0]=one
array[1]=two and four
$ unset entry index
# Coming back to motivation: GOOD code!
$ files=(~/*.jpg); cp "${files[@]}" /backups/
```

Deleting arrays or array elements

- To delete an element of an array, pass it to the `unset` builtin

```
$ array=({a..e}); unset 'array[2]' 'array[3]'
$ for index in "${!array[@]}"; do
>     echo "array[${index}]=${array[index]}"
> done; unset 'index'
array[0]=a
array[1]=b
array[4]=e
```

- If you pass the name of an array to `unset`, the whole array will be unset!

```
$ array=({a..c})
$ for index in "${!array[@]}"; do
>     echo "array[${index}]=${array[index]}"
> done; unset 'index'
array[0]=a
array[1]=b
array[2]=c
$ unset array
$ echo "_${array[@]}_ -> ${#array[@]} entries"
__ -> 0 entries
```

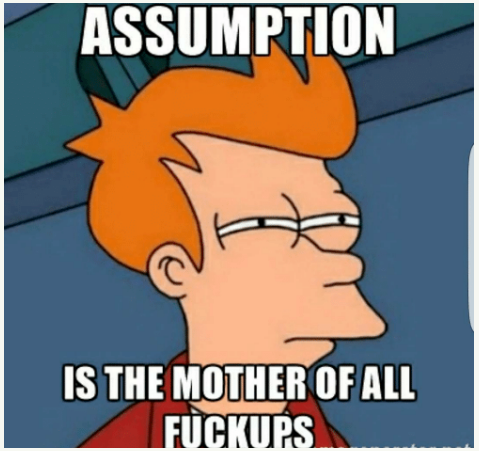
Deleting arrays or array elements

Do not forget to **quote your variable** to avoid file pattern matching!

```
$ a=(file{1..3})
$ ls
a0   Day_1.tex   Day_1.pdf
$ a0='Hello'
$ printf '%s\n' "${a[@]}"
file1
file2
file3
$ echo "${a0}"
Hello
$ unset a[0] # <--- BAD, BAD, BAD CODE!
$ printf '%s\n' "${a[@]}"
file1   # Wait! I unset the first element, didn't I?
file2
file3
$ echo "_${a0}_"
__                  # Is it clear what happened?
```


In doubt, always quote the variable passed to `unset`. Using either the `-v` or `-f` option is also encouraged!

Sparse arrays



Arrays might be sparse!

- Don't assume that your indices are sequential
- If the index values matter, always iterate over the indices { instead of making assumptions about them... }
- If you loop over the values instead, don't assume anything about indices
- In particular, don't assume that just because you're currently in the first iteration of your loop, that you must be on index 0...

If you need a non-sparse array, just make it such!

```
$ array=("${array[@]}")
```

Associative arrays

Since Bash v4.0 (2009)

- An associative array is a map of strings to strings
- It has to be **explicitly declared** as such via `declare -A`
- Since keys are strings, they might contain spaces!
  ⇒ Iterating over keys requires quotes: `"${!array[@]}"`

```
$ declare -A dict
$ dict[Apartment]="Die Wohnung"
$ declare -p dict
declare -A dict='([Apartment]="Die Wohnung" )'
$ dict[Water]="Das Wasser"
$ for key in "${!dict[@]}"; do
>     printf 'EN: %20s -> DE: %s\n' "${key}" "${dict[$key]}"
> done
EN:                 Water -> DE: Das Wasser
EN:             Apartment -> DE: Die Wohnung
$ echo "${#dict[@]}"; unset dict
2
$ echo "${#dict[@]}"
0
```

Associative arrays

Since Bash v4.0 (2009)

- An associative array is a map of strings to strings
- It has to be **explicitly declared** as such via `declare -A`
- Since keys are strings, they might contain spaces!
  ⇒ Iterating over keys requires quotes: `"${!array[@]}"`

```
$ declare -A dict
$ dict[Die Wohnung]="Apartment"
$ dict[Das Wasser]="Water"
$ for key in ${!dict[@]}; do    # AAAAAARGH!
>     printf 'DE: %20s -> EN: _%s_\n' "${key}" "${dict[$key]}"
> done
DE:                  Die -> EN: __
DE:              Wohnung -> EN: __
DE:                  Das -> EN: __
DE:               Wasser -> EN: __
$ echo "${#dict[@]}"; unset -v 'dict[Das Wasser]'
2
$ echo "${#dict[@]}"; unset -v 'dict'
1
```

Associative arrays: Remarks

Since Bash v4.0 (2009)

- The order of both keys and elements of an associative array is unpredictable
  ⇒ They are **not** well suited to store lists that need to be processed in order
- If you use a parameter as key of an associative array, you must use the $ sign
  ⇒ Indeed, it makes sense: The name of the parameter might simply be a valid key

```
1 $ declare -A array
2 $ index=1; for entry in one two three four five six; do
3 >     array[${entry}]=$((index++))
4 > done
5 $ echo "${array[@]}"
6 4 1 5 6 2 3
7 $ echo "${!array[@]}"
8 four one five six two three
9 $ unset array entry index
```

Associative arrays: Remarks

Since Bash v4.0 (2009)

- The order of both keys and elements of an associative array is unpredictable
  ⇒ They are **not** well suited to store lists that need to be processed in order
- If you use a parameter as key of an associative array, you must use the $ sign
  ⇒ Indeed, it makes sense: The name of the parameter might simply be a valid key

```
$ indexed=("one" "two")  index=0  key="foo"
$ declare -A associative=([foo]=bar [alpha]=omega)
$ echo "${indexed[$index]}"
one
$ echo "${indexed[index]}"
one
$ echo "${indexed[index + 1]}"
two
$ echo "${associative[$key]}"
bar
$ echo "_${associative[key]}_"
__
$ echo "_${associative[key + 1]}_"
__
$ unset indexed associative index key
```

The power of parameter expansion on arrays

- Arrays are very flexible, because they are well integrated with the other shell expansions
- Any parameter expansion can be carried out on individual array elements
- Parameter expansions can equally well apply to an entire array!
- To use parameter-expansion manipulations on all array entries, use `[@]` and `[*]`
- It is critical that these special expansions are properly quoted

```
$ array=(alpha beta gamma)
$ echo "${array[-1]:2:2}"   # Substring of last entry
mm
$ echo "${array[@]#a}"      # Chop 'a' from the beginning
lpha beta gamma
$ echo "${array[@]%a}"      # Chop 'a' from the end
alph bet gamm
$ echo "${array[@]//a/_}"   # Substitute all 'a' by '_'
_lph_ bet_ g_mm_
$ echo "${array[@]/#a/_}"   # Substitute 'a' by '_' at start
_lpha beta gamma
$ echo "${array[@]/%a/_}"   # Substitute 'a' by '_' at end
alph_ bet_ gamm_
```

The power of parameter expansion on arrays

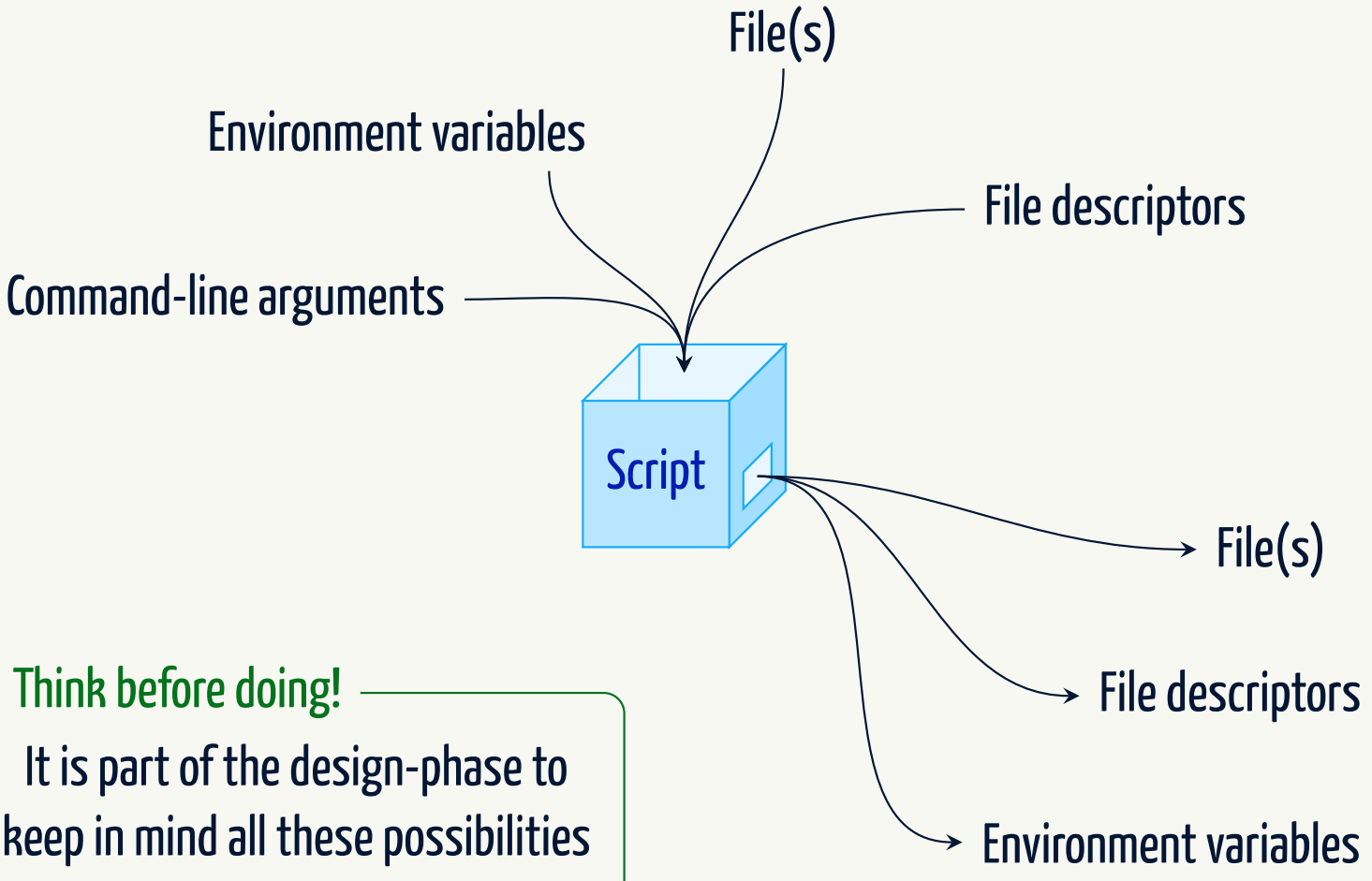
- Arrays are very flexible, because they are well integrated with the other shell expansions
- Any parameter expansion can be carried out on individual array elements
- Parameter expansions can equally well apply to an entire array!
- To use parameter-expansion manipulations on all array entries, use `[@]` and `[*]`
- It is critical that these special expansions are properly quoted

```
$ echo "${array[@]/#/prefix_}"  # Add prefix to all entries
prefix_alpha prefix_beta prefix_gamma
$ echo "${array[@]/%/_suffix}"  # Add suffix to all entries
alpha_suffix beta_suffix gamma_suffix
$ echo "${array[@]^^}"          # Capitalise all characters
ALPHA BETA GAMMA
$ echo "${array[3]?Third entry is unset}"
bash: array[3]: Third entry is unset
$ echo "${array[3]=delta}"
delta
$ echo "${array[@]^}"
Alpha Beta Gamma Delta
$ unset array
```

Input and output

The Krisuvikurberg cliffs at the sunset

The Bash script flow



Think before doing!

It is part of the design-phase to keep in mind all these possibilities and take advantage of them

Command-line parameters

- They are accessible via `$1`, `$2`, etc.
- After the 9th one, you need curly braces: `${10}`, `${11}`, etc.
- It is possible to refer to all of them via `$@` and `$*`
- When you refer to all of them, especially to pass them over, use `"$@"`
  → the double quotes are crucial to preserve the parameters without splitting them!
- The `shift` bulit-in is remarkably handy when parsing command-line parameters
  → it destroys `$1` and it maps `$2` into `$1`, `$3` into `$2` and so on

```
$ set -- Hello my "nice world"
$ printf '%s\n' "$@"
Hello
my
nice world
$ printf '%s\n' $@   # <-- Probably, AAAAARGH!
Hello
my
nice
world
```

Command-line parameters

- They are accessible via `$1`, `$2`, etc.
- After the 9th one, you need curly braces: `${10}`, `${11}`, etc.
- It is possible to refer to all of them via `$@` and `$*`
- When you refer to all of them, especially to pass them over, use `"$@"`
  → the double quotes are crucial to preserve the parameters without splitting them!
- The `shift` bulit-in is remarkably handy when parsing command-line parameters
  → it destroys `$1` and it maps `$2` into `$1`, `$3` into `$2` and so on

```
$ echo "$1"; shift
Hello
$ echo "$1"; shift
my
$ echo "$1"; shift
nice world
$ echo _"$1"_
__
$ set -- Hello my "nice world"; shift 2; echo "$1"
nice world
```

Environment variables (I)

- Every program inherits certain information from its parent process { resources, privileges and restrictions }
- One of those resources is a set of variables called Environment Variables
- Traditionally, environment variables have names that are all capital letters, such as `PATH`
- When you run a command in Bash, you have the option of specifying a temporary environment change which only takes effect for the duration of that command
- This is done by putting `VAR=value` in front of the command

```
1 $ ls /tpm
2 ls: cannot access '/tpm': No such file or directory
3 $ LANGUAGE=de_DE.utf-8 ls /tpm  # 🔗Read more about LANGUAGE
4 ls: Zugriff auf '/tpm' nicht möglich: Datei oder
      Verzeichnis nicht gefunden
5 $ VERBOSE=1 make
```

Good parctice

Don't use all-capital variable names in your scripts, unless they are environment variables. Use lower-case or mixed-case variable names, to avoid accidents.

Environment variables (II)

- In a script, you can use environment variables just like any other variable

```
if [[ ${EDITOR} ]]; then
    ${EDITOR}
else
    emacs
fi
```

- To change the environment for your child processes to inherit, use the `export` builtin

```
export PATH=${HOME}/.local/bin:${PATH}
```

Remember!

The tricky part here is that your environment changes are **only inherited by your descendants**. You can't change the environment of a program that is already running, or of a program that you don't run.

...mmmh, and if I needed it?

Environment variables (III)

- Sourcing a script, will execute it in the current environment/shell

```
$ cat script.bash
#!/bin/sh
cd /tmp
$ pwd; ./script.bash; pwd
/home/sciarra/Documents
/home/sciarra/Documents
$ pwd; source script.bash; pwd # 'source' as '.'
/home/sciarra/Documents
/tmp
```

Indeed, this is what you do e.g. when you add code in the `${HOME}/.bashrc` file

Splitting a large script in several files

Although a script should not be huge, it is important sometimes to split it into pieces for a handier development. This can be done using the `source` builtin, delegating to the main (executable) script to source all secondary files. We will come back to this point when we introduce functions.

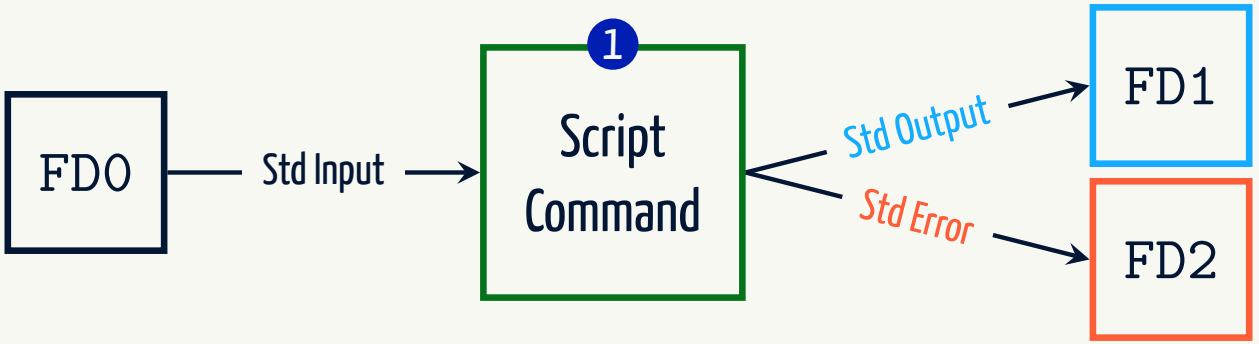
File descriptors and redirection

Horses

File descriptors: a graphical overview

By default

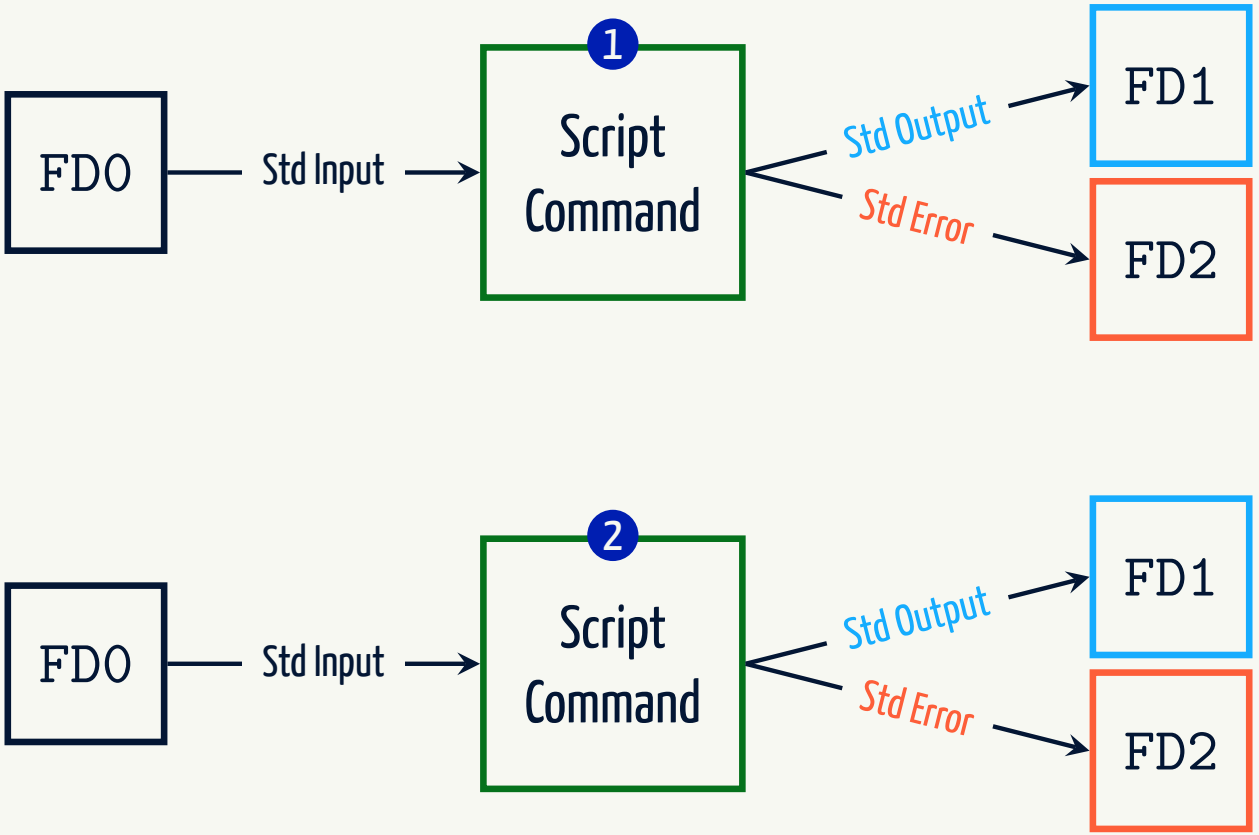
Every new process starts with **three** open file descriptors



File descriptors: a graphical overview

By default

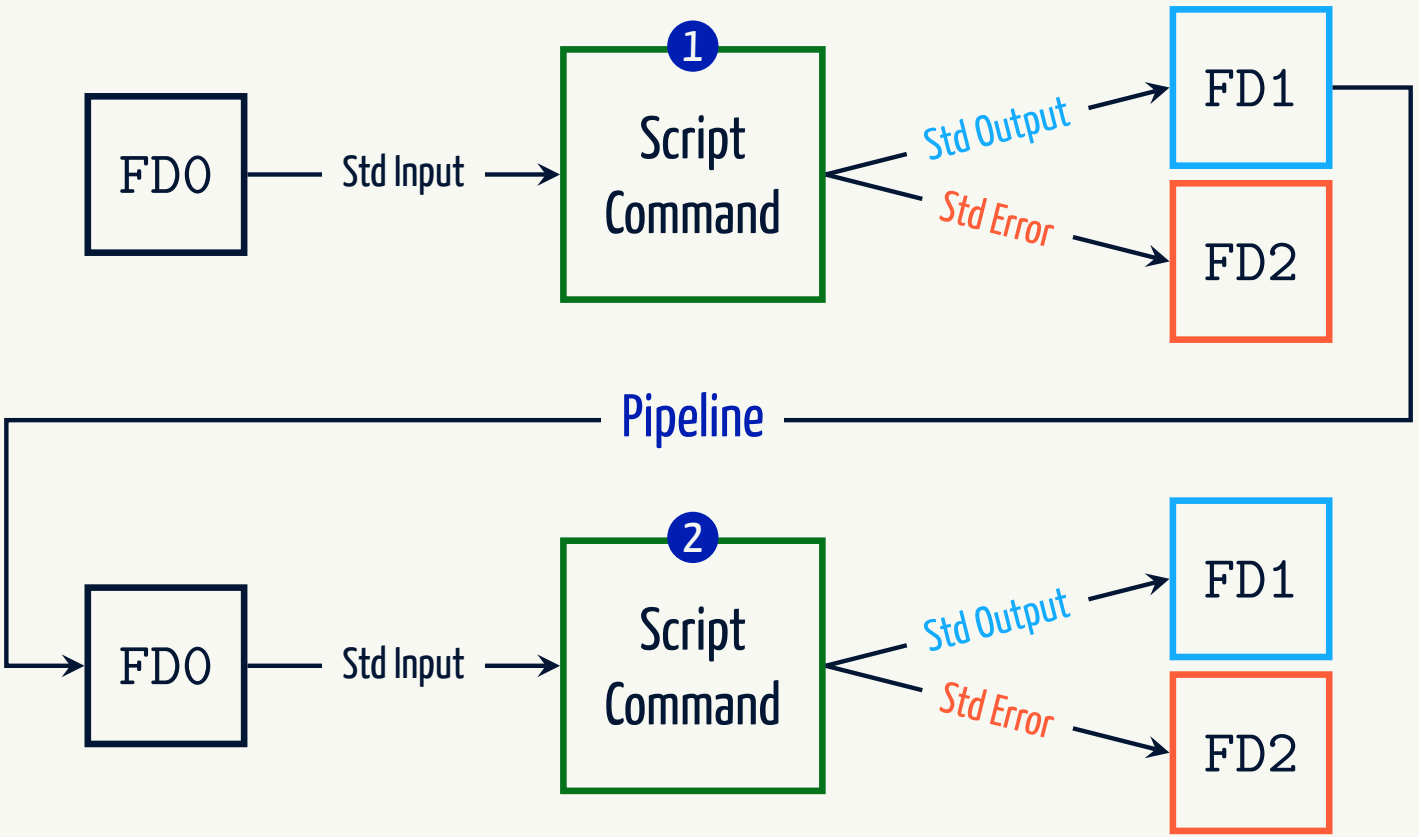
Every new process starts with **three** open file descriptors



File descriptors: a graphical overview

By default

Every new process starts with **three** open file descriptors



File descriptors

In an interactive shell, or in a script running in a terminal,

0. the **Standard Input** is how Bash sees the characters you type on your keyboard;
1. the **Standard Output** is where the program sends most of its normal information;
2. the **Standard Error** is where the program sends its error messages.

> **Good practice**
>
> Remember that when you create scripts, you should send your custom error messages to the standard error FD. This is a convention and it is very convenient when applications follow the convention. **As such, so should you!**

In the following we will learn how to

- redirect standard {input, error, output} of {single,multiple} commands;
- open and close new file descriptors;
- use and not abuse the pipeline.

Input/Output Redirection: Some examples

Redirection

The practice of changing a FD to read its input from, or send its output to, a different location

```
# Standard OUTPUT redirection
$ ls; echo 'Pi*10^7s approximates at 0.38% a year'
Pi*10^7s approximates at 0.38% a year
$ echo 'Pi*10^7s approximates at 0.38% a year' > Quote.txt
$ ls; cat Quote.txt
Quote.txt
Pi*10^7s approximates at 0.38% a year
$ echo '82000 is a super cool number!' > Quote.txt
$ cat Quote.txt
82000 is a super cool number! # We lost the file content!
$ echo 'Pi*10^7s approximates at 0.38% a year' >  Quote.txt
$ echo '82000 is a super cool number!'          >> Quote.txt
$ cat Quote.txt
Pi*10^7s approximates at 0.38% a year
82000 is a super cool number!
$ wc -l Quote.txt
2 Quote.txt
$ man wc # What happens if you do not provide a file to wc?
```

Input/Output Redirection: Some examples

Redirection

The practice of changing a FD to read its input from, or send its output to, a different location

```
# Standard INPUT redirection
$ wc -l < Quote.txt
2
# Standard ERROR redirection
$ rm Quote.ttx
rm: cannot remove 'Quote.ttx': No such file or directory
$ rm Quote.ttx > Error.log
rm: cannot remove 'Quote.ttx': No such file or directory
$ ls; wc -l Error.log
Error.log   Quote.txt
0 Error.log
$ rm Quote.ttx 2> Error.log
$ wc -l Error.log
1 Error.log
$ rm Quote.ttx 2>> Error.log
$ cat Error.log
rm: cannot remove 'Quote.ttx': No such file or directory
rm: cannot remove 'Quote.ttx': No such file or directory
```

Input/Output Redirection: Some examples

Redirection

The practice of changing a FD to read its input from, or send its output to, a different location

```
# Use of the pipeline
$ cat Error.log Quote.txt | sort
82000 is a super cool number!
Pi*10^7s approximates at 0.38% a year
rm: cannot remove 'Quote.ttx': No such file or directory
rm: cannot remove 'Quote.ttx': No such file or directory
$ cat Error.log Quote.txt | sort | uniq
82000 is a super cool number!
Pi*10^7s approximates at 0.38% a year
rm: cannot remove 'Quote.ttx': No such file or directory
# BAD CODE (Abuse of the pipeline)
$ cat Quote.txt | grep 'cool'
82000 is a super cool number!
# GOOD CODE: grep 'cool' Quote.txt
# BAD Bash CODE (Abuse of the pipeline)
$ echo "2*5.3" | bc -l
10.6
# You should use the Herestrings syntax we'll discuss later
```

Input/Output Redirection: The theory

- I/O redirection is done **before** the command is executed!
- Redirections are processed in the order they appear, **from left to right**
- In the following slides `[n]` refers to an optional integer, whose default is
  - `0` if the redirection operator is `<` or `<>`
  - `1` if the redirection operator is `>` or `>>`
- The `word` following the redirection operator in the following slides is subjected to*
  - brace expansion
  - tilde expansion
  - parameter expansion
  - command substitution
  - arithmetic expansion
  - quote removal
  - filename expansion
  - word splitting

  If it expands to more than one word, Bash reports an error.

* For heredocs and herestrings a different rule applies: Refer to the Bash manual for more details.

I/O Redirection: The syntax

Redirecting Input: `[n]<word`

The file resulting from the expansion of `word` is opened for reading on FD `n`

Redirecting Output: `[n]>word`

The file resulting from the expansion of `word` is opened for writing on FD `n`

{ If the file does not exist it is created; if it does exist it is **truncated to zero size** }

Appending Output: `[n]>>word`

The file resulting from the expansion of `word` is opened for appending on FD `n`

Duplicating FDs: `[n]<&number` and `[n]>&number`

The file descriptor denoted by `n` is made to be a copy of file descriptor `number`

If `number` does not specify an open file descriptor, a redirection error occurs

Closing FDs: `[n]<&-` and `[n]>&-`

The file descriptor denoted by `n` is closed

Moving FDs: `[n]<&number-` and `[n]>&number-`

The FD `number` is moved to FD `n` and FD `number` is then closed

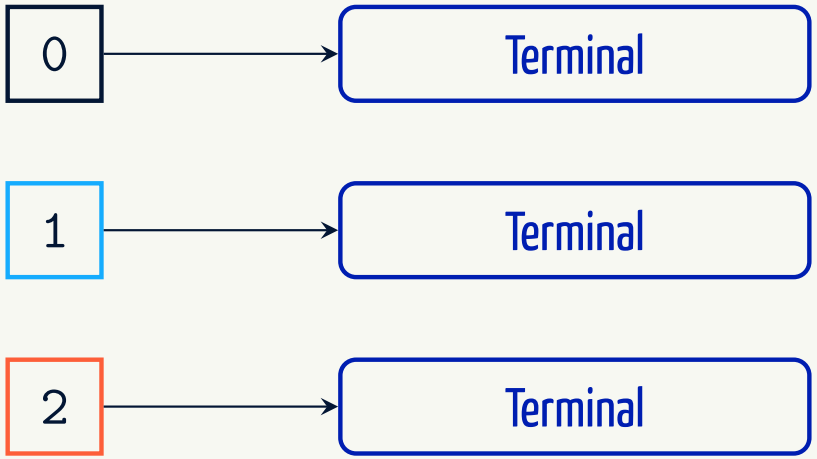
I/O FDs: `[n]<>word`

The file resulting form `word` is opened for both reading and writing on FD `n`

{ If the file does not exist, it is created }

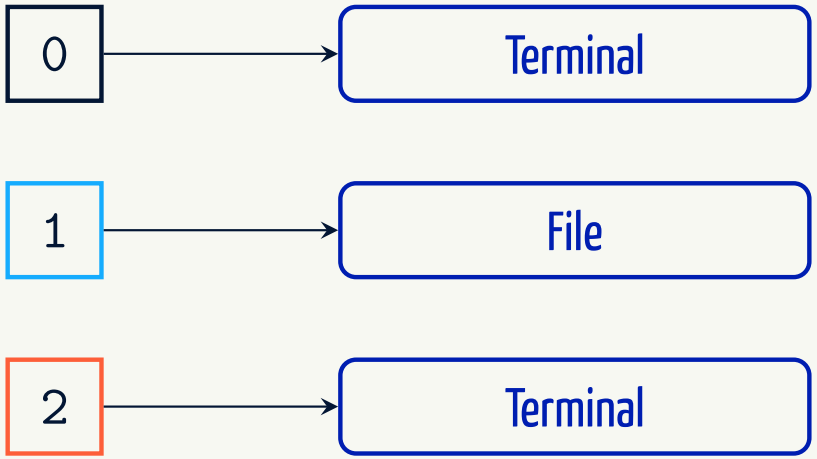
Few more basic examples

```
# Standard prcess situation
$ command
```



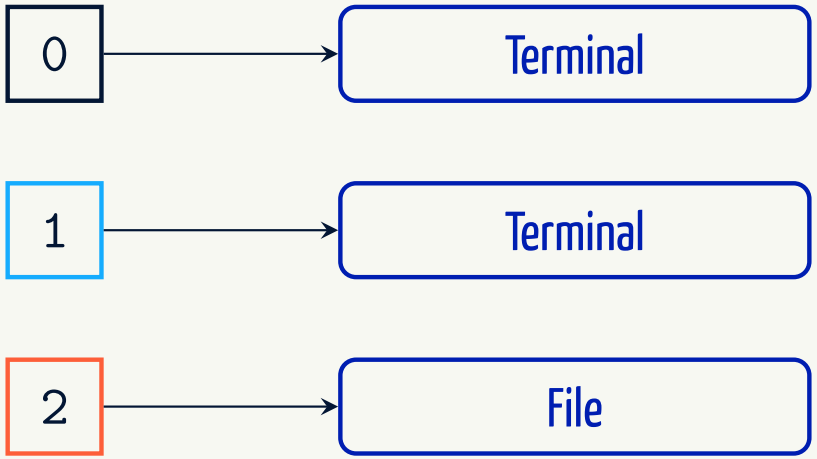
Few more basic examples

```
# Redirect the standard output of a command to a file
$ command > File
```



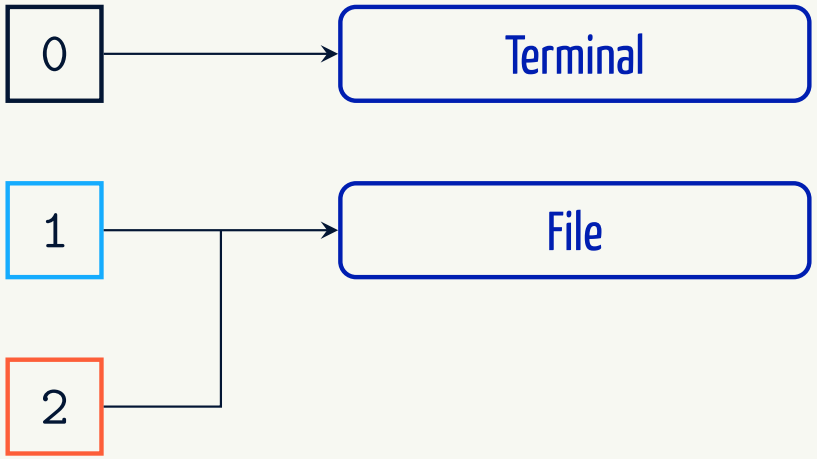
Few more basic examples

```
# Redirect the standard error of a command to a file
$ command 2> File
```



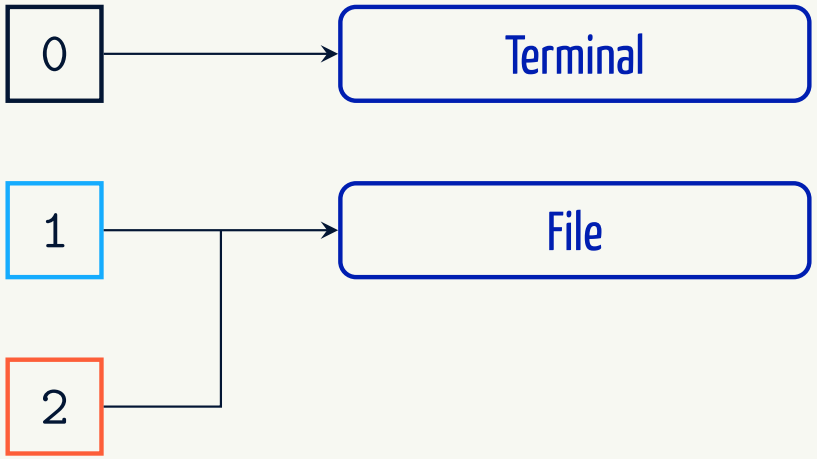
Few more basic examples

```
# Redirect both std output and std error to a file
$ command &> File
# Equivalent to
$ command > File 2>&1
# WRONG CODE! Explanation later
# command > File 2> File
```



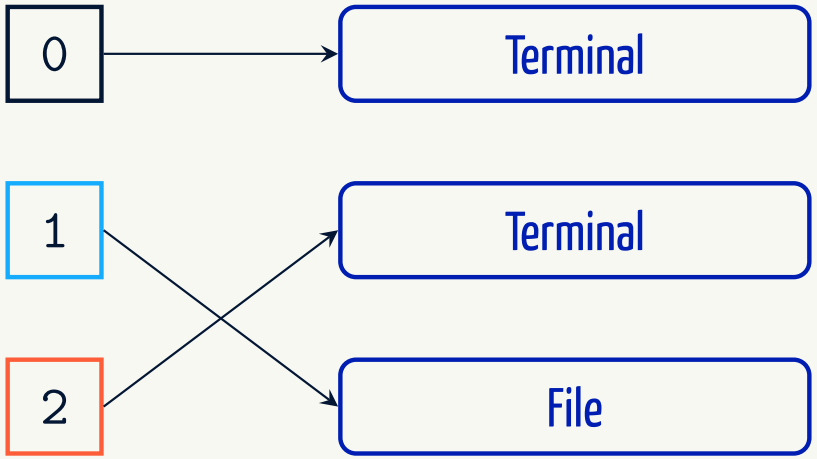
Few more basic examples

```
# The order of the redirections matters!
$ command > File 2>&1
# command 2>&1 > File
```



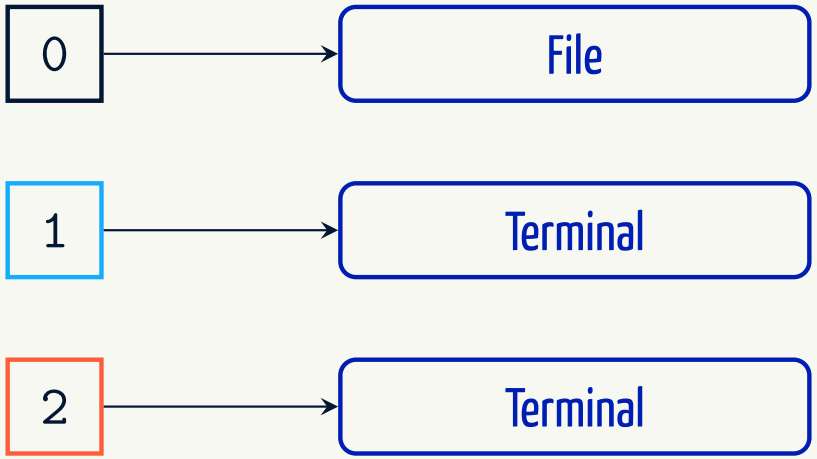
Few more basic examples

```
# The order of the redirections matters!
# command > File 2>&1
$ command 2>&1 > File
```



Few more basic examples

```
# Redirect the std input of a command to a file
$ command < File
```



The `exec` builtin

> **`exec …[command [arguments]]`**
>
> […] If no command is specified, redirections may be used to affect the current shell environment […]
>
> — Bash manual

```
# Redirect both std output and std error to "log.txt"
$ exec > log.txt 2>&1
# ALL output including stderr now goes into "log.txt"
```

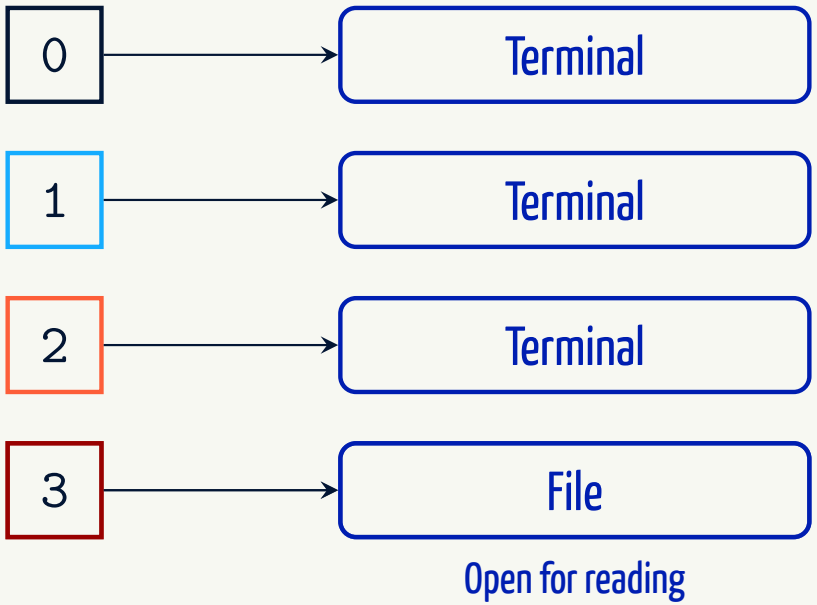
Otherwise, for local changes, command grouping helps:

```
# Redirect both std output and std error to "messages.log"
{
    date
    # some other commands
    echo '...done!'
} > messages.log 2>&1
```


To check the open file descriptors you can use: `ls -l /proc/$$/fd`

Few more advanced examples

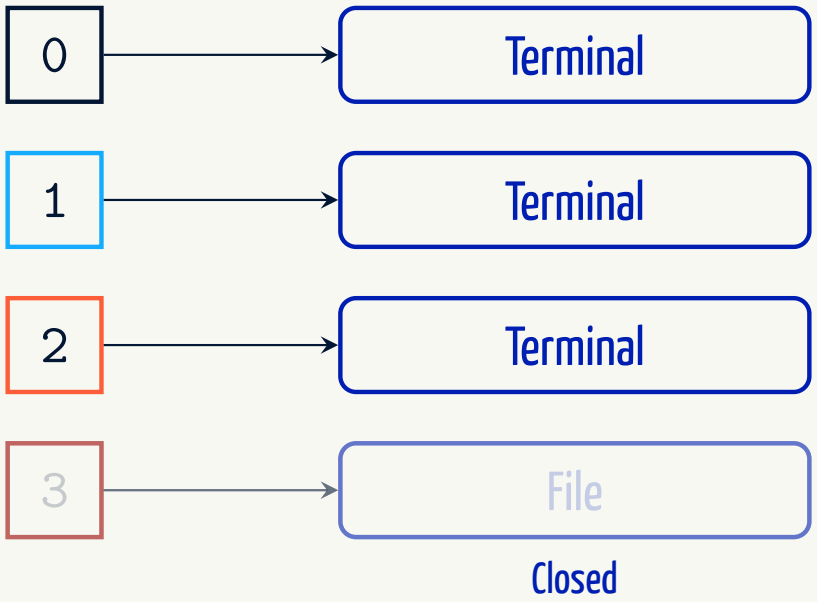
```
# Open a file for reading using a custom file descriptor
$ exec 3< File
# grep "foo" <&3 # NOTE: You can't rewind a fd in Bash!
# ...                   Close and open it again in case
```



File descriptor numbers range from 0 to 255.

Few more advanced examples

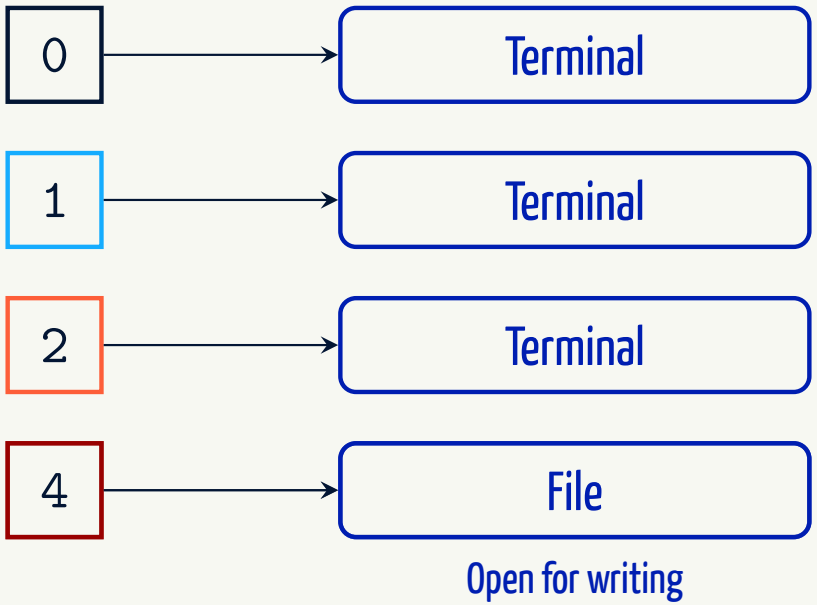
```
# Open a file for reading using a custom file descriptor
$ exec 3< File
# grep "foo" <&3 # NOTE: You can't rewind a fd in Bash!
# ...                    Close and open it again in case
$ exec 3<&-
```



File descriptor numbers range from 0 to 255.

Few more advanced examples

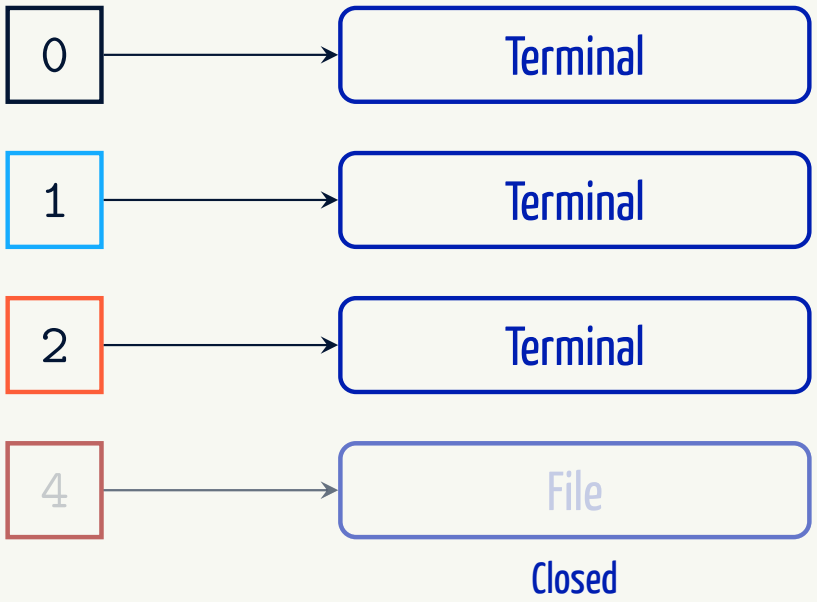
```
# Open a file for writing using a custom file descriptor
$ exec 4> File
# echo "foo" >&4
# ...
```



File descriptor numbers range from 0 to 255.

Few more advanced examples

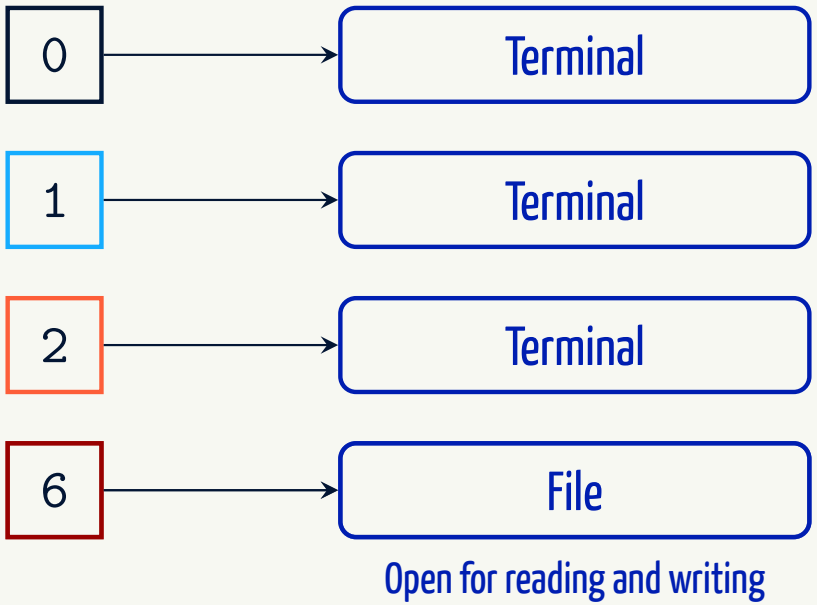
```
# Open a file for writing using a custom file descriptor
$ exec 4> File
# echo "foo" >&4
# ...
$ exec 4>&-
```



File descriptor numbers range from 0 to 255.

Few more advanced examples

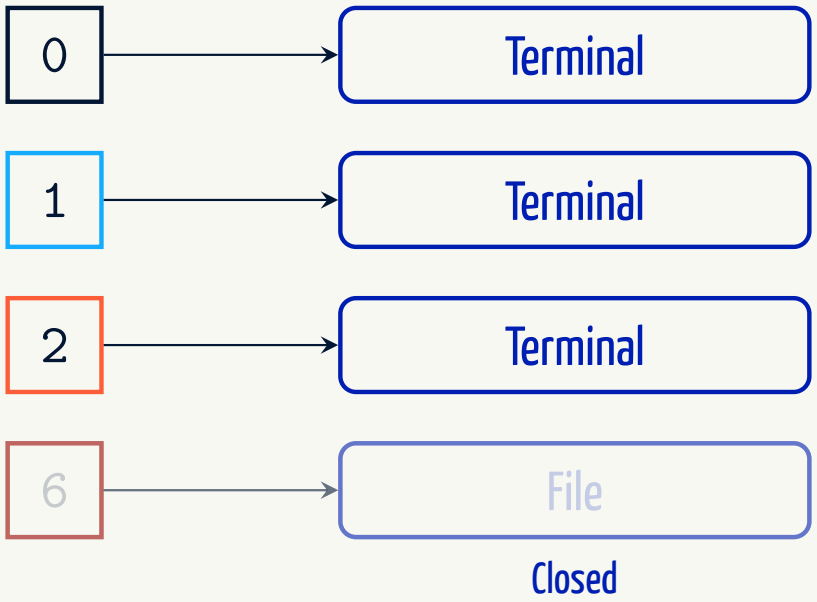
```
$ echo "foo bar" > File
# Open a file both for writing and reading with a custom FD
$ exec 6<> File
$ read -n 3 var <&6  # read the first 3 characters from FD 6
$ echo $var
$ echo -n + >&6      # write "+" at 4th position
```



File descriptor numbers range from 0 to 255.

Few more advanced examples

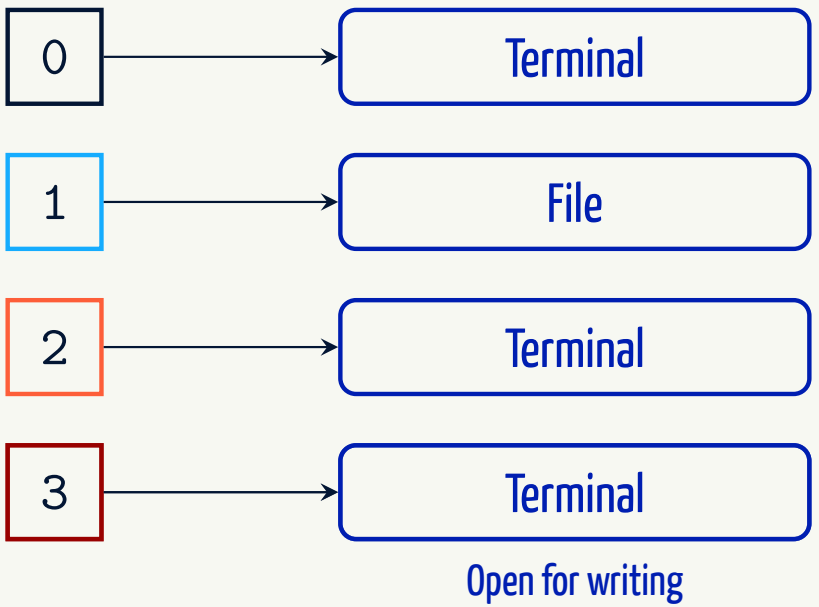
```
$ echo "foo bar" > File
# Open a file both for writing and reading with a custom FD
$ exec 6<> File
$ read -n 3 var <&6  # read the first 3 characters from FD 6
$ echo $var
$ echo -n + >&6      # write "+" at 4th position
$ exec 6>&-          # close FD 6
$ cat File
```



File descriptor numbers range from 0 to 255.

Few more advanced examples

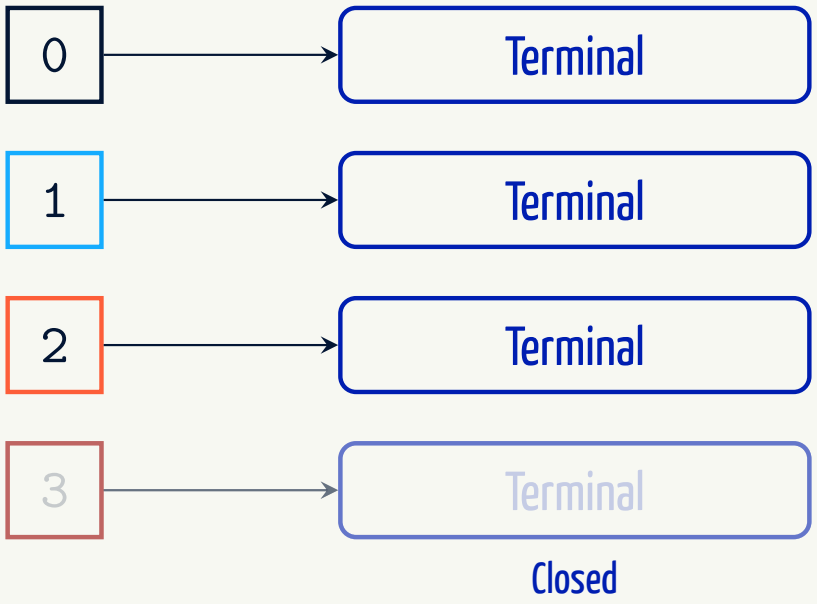
```
# Redirect standard output temporarily to file
$ exec 3>&1 1> File
$ date
$ echo "This goes to File without redirection!"
$ date
```



File descriptor numbers range from 0 to 255.

Few more advanced examples

```
# Redirect standard output temporarily to file
$ exec 3>&1 1> File
$ date
$ echo "This goes to File without redirection!"
$ date
$ exec 1>&3 3>&-
$ echo "I see this in the terminal again!"
$ cat File
```



File descriptor numbers range from 0 to 255.

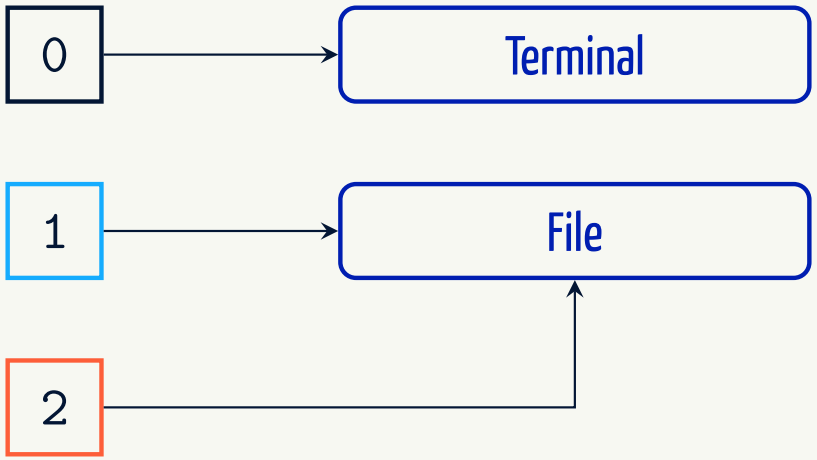
Redirecting standard output and error to the same place

Why does the naive way fail?

```
$ command > File 2> File    # WRONG CODE!
```

- We have created a very bad condition here
- Two FDs that both point to the same file, independently of each other, is a **bad idea**
- The results of this are not well-defined
- Some information written via one FD may clobber information written through the other FD

  { Depending on how the operating system handles FDs }

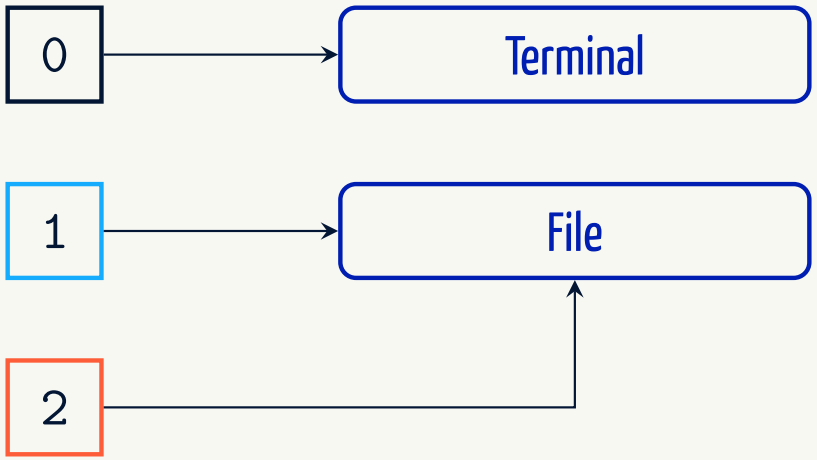


Redirecting standard output and error to the same place

Why does the naive way fail?

```
$ command > File 2> File    # WRONG CODE!
```

```
$ echo "I am a very proud sentence with a lot of words in
    it, all for you." > File
$ grep proud File 'not a file' > proud.log 2> proud.log
$ cat proud.log
grep: not a file: No such file or directory
f words in it, all for you.
```



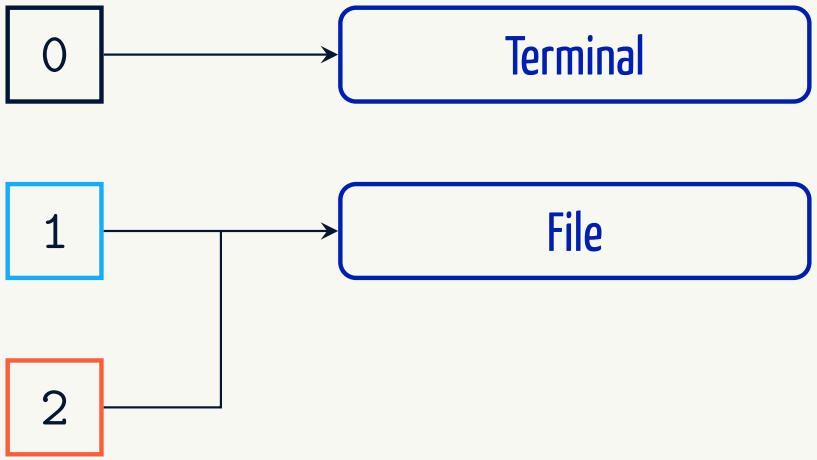
Redirecting standard output and error to the same place

Why does the naive way fail?

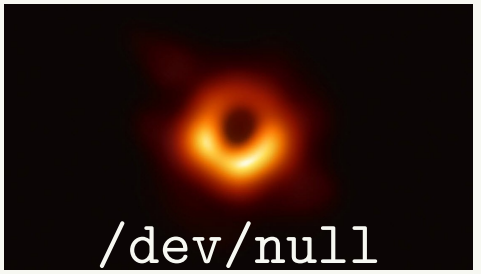
```
$ command > File 2> File    # WRONG CODE!
```

How should I fix this?

```
$ command  > File 2>&1    # GOOD CODE!
$ command &> File         # ALSO GOOD (Bash specific)
```



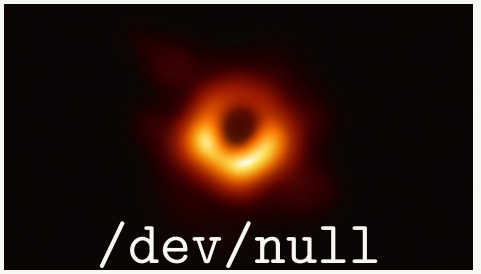
A kind of black hole in the operative system




```
$ echo 'Now you see me!'
Now you see me!
$ echo 'Now you do not see me!'  > /dev/null
$ echo 'Now you do not see me!' >> /dev/null
$ rm 'Who cares that this does not exists' 2>  /dev/null
$ rm 'Who cares that this does not exists' 2>> /dev/null
```

- The file `/dev/null` is always empty, no matter what you write to it or read from it
- As such, when we write our (error) messages to it, they just disappear
- The `/dev/null` file remains as empty as ever before
- That's because it is not a normal file: It is a virtual device
- Redirecting `/dev/null` to **standard input** will give an immediate EOF to any read call

A kind of black hole in the operative system



Do not be too clever!

Suppressing the standard error might not be a so smart idea...

```
$ echo 'Now you see me!'
Now you see me!
$ echo 'Now you do not see me!'  > /dev/null
$ echo 'Now you do not see me!' >> /dev/null
$ rm 'Who cares that this does not exists' 2>  /dev/null
$ rm 'Who cares that this does not exists' 2>> /dev/null
```

- The file `/dev/null` is always empty, no matter what you write to it or read from it
- As such, when we write our (error) messages to it, they just disappear
- The `/dev/null` file remains as empty as ever before
- That's because it is not a normal file: It is a virtual device
- Redirecting `/dev/null` to **standard input** will give an immediate EOF to any read call

Here Documents (I)

```
command [n]<<[-]WORD
    # here document
WORD
```

- Heredocs are useful to embed **short** blocks of multi-line data inside your script
  { Embedding larger blocks is bad practice! }
- In a Heredoc, you need to choose a `WORD` to act as a sentinel
- It can be any word: Choose one that won't appear in your data set
- All the lines that follow the first instance of the sentinel, up to the second instance, become the standard input for the `command`
- The second instance of the sentinel word has to be at the beginning of a line all by itself

Good practice

You should keep your logic { your code } and your input { your data } separated, preferably in different files, unless it i s a small data set!

Here Documents (II)

1. Standard form

```
command <<WORD
    # here document subjected to parameter expansion,
    # command substitution, and arithmetic expansion
WORD
```

2. Quoting (part of) the sentinel word

```
command <<'WORD'
    # the lines in the here-document are not expanded
WORD
```

3. Adding a dash before the sentinel word to indent code

```
command <<-WORD
    # Leading tabs (NOT spaces!) are removed
WORD
```

4. Combining the previous two forms is clearly allowed

No parameter and variable expansion, command substitution, arithmetic expansion, or filename expansion is performed on `WORD`.

Here Documents (III): Examples

```
# Standard form
$ if [[ $(date +'%A') != S* ]]; then
>     cat <<END
>         Ouch, it is not weekend: $(date +'%A')
> END
> fi
       Ouch, it is not weekend: Tuesday
# Removing leading tabs
$ cat <<-END
>     abc seems to start with an a! # TAB before abc
> END                               # (CTRL-v TAB)
abc seems to start with an a!
# Avoiding expansion
$ cat <<-'SENTINEL'
>     My home is ${HOME}            # TAB before My
> SENTINEL
My home is ${HOME}
```

Here Documents (III): Examples

```
# Standard form
$ if [[ $(date +'%A') != S* ]]; then
>     cat <<END
>         Ouch, it is not weekend: $(date +'%A')
> END
> fi
       Ouch, it is not weekend: Tuesday
# Removing leading tabs
$ cat <<-END
>     abc seems to start with an a! # TAB before abc
> END                                # (CTRL-v TAB)
abc seems to start with an a!
# Avoiding expansion
$ cat <<-'SENTINEL'
>     My home is ${HOME}             # TAB before My
> SENTINEL
My home is ${HOME}
```

```
cat <<EOF
usage: foobar [-x] [-v] [-z] [file ...]
  A short explanation of the operation goes here.
  It might be few lines long, but should not be excessive.
EOF
```

Here Strings

```
command [n]<<< STRING
```

- Herestrings are shorter, less intrusive and overall more convenient than Heredoc

  { However, they are not portable to the Bourne shell }
- The `STRING` undergoes:
  - tilde expansion
  - parameter and variable expansion
  - command substitution
  - arithmetic expansion
  - quote removal
- Pathname expansion and word splitting are **not** performed
- The result is supplied as a **single** string, with a newline appended, to the `command` on its standard input (or file descriptor `n` if specified)

Good practice

Prefer Herestrings to pipes whenever possible!

Here Strings

```
command [n]<<< STRING
```

- Herestrings are shorter, less intrusive and overall more convenient than Heredoc

  { However, they are not portable to the Bourne shell }

- The `STRING` undergoes:
  - tilde expansion
  - parameter and variable expansion
  - command substitution
  - arithmetic expansion
  - quote removal

Do not forget quotes, especially if the string contains spaces!

- Pathname expansion and word splitting are **not** performed
- The result is supplied as a **single** string, with a newline appended, to the `command` on its standard input (or file descriptor `n` if specified)

Good practice

Prefer Herestrings to pipes whenever possible!

Here Strings

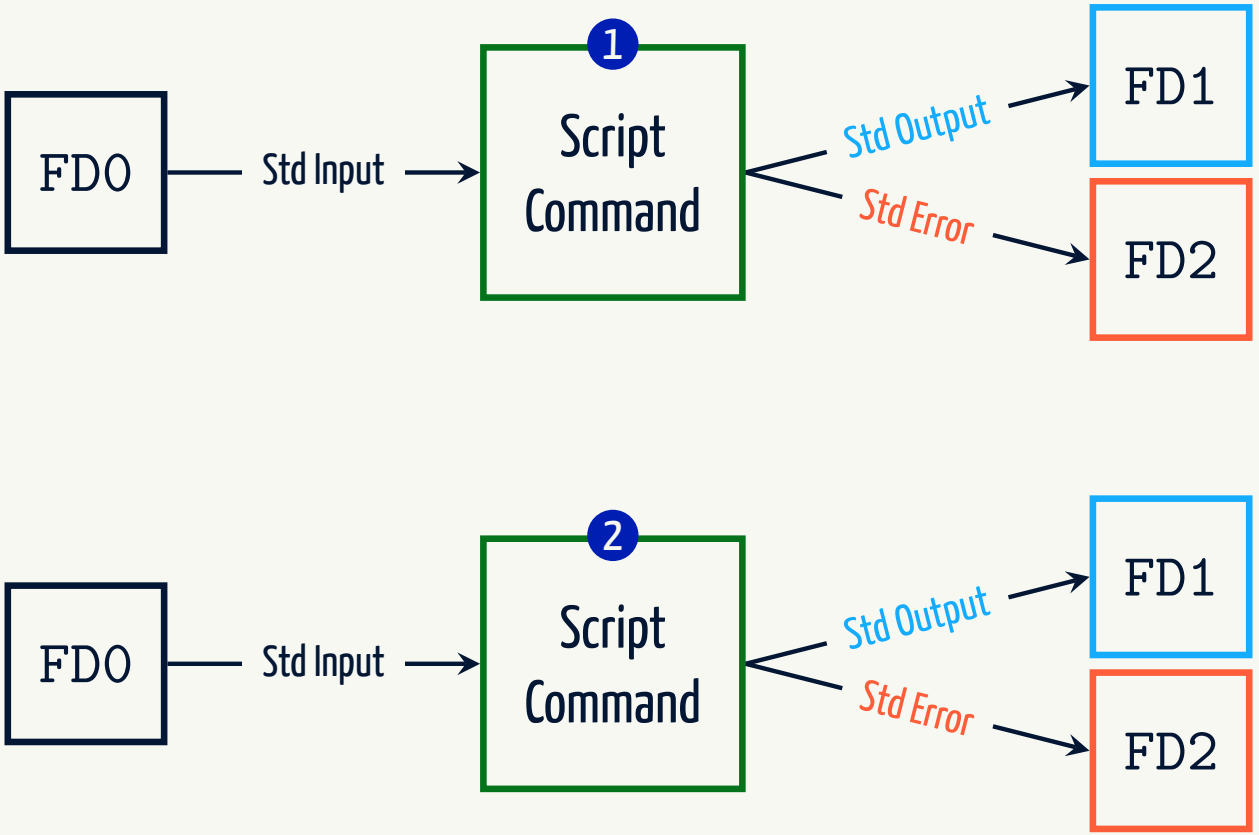
```
command [n]<<< STRING
```

```
$ grep --color=auto '[aeiou]' <<< "Clark Kent"
Clark Kent
$ bc <<< "10/4"
2
$ bc -l <<< "10/4"
2.50000000000000000000
$ bc <<< "obase=16; ibase=$(date +'%w'); 11000000"
C0
$ bc <<< "ibase=2; obase=10000; 11000000"
C0
$ wc -c <<< "Hello"
6
$ ls
Day_1.pdf   Day_2.pdf
$ echo *
Day_1.pdf Day_2.pdf
$ wc -c <<< *           # No filename expansion!
2                       # Why 2 characters?
```

File descriptors: a graphical overview

By default

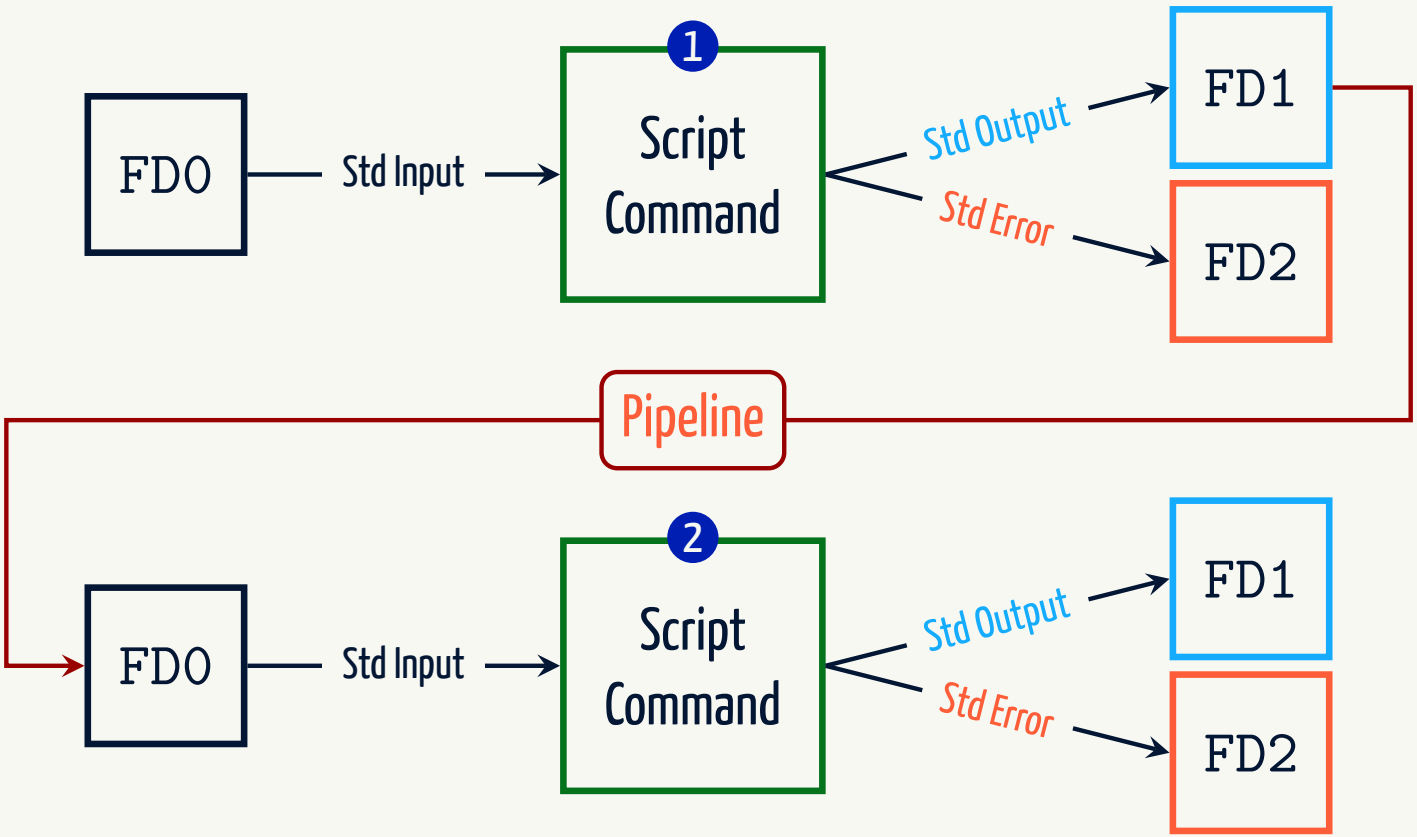
Every new process starts with **three** open file descriptors



File descriptors: a graphical overview

By default

Every new process starts with **three** open file descriptors



Pipes

```
command | command
```

- The pipe is created using the `|` operator between two commands
- The former `command`'s std output is connected to the latter `command`'s std input
- This connection is performed before any redirection specified by the commands
- Pipes are widely used as a means of post-processing application output
- The pipe operator creates a subshell environment for each command! { More on subshells tomorrow }

```
1 $ message='Test'
2 $ echo 'Salut, le monde!' | read message
3 $ echo "The message is: $message"
4 The message is: Test
5 $ echo 'Salut, le monde!' | { read message;
6 > echo "The message is: $message"; }
7 The message is: Salut, le monde!
8 $ echo "The message is: $message"
9 The message is: Test
```

Pipes

```
command | command
```

- The pipe is created using the `|` operator between two commands
- The former `command`'s std output is connected to the latter `command`'s std input
- This connection is performed before any redirection specified by the commands
- Pipes are widely used as a means of post-processing application output
- The pipe operator creates a subshell environment for each command! { More on subshells tomorrow }

```
command 2>&1 |   command
command      |& command
```

- `|&` also connects the first `command`'s std error to the second `command`'s std input
- It is shorthand for `2>&1 |`
- This implicit redirection is performed after any redirection specified by the commands

Pipes: A simple example

```
# Finding the maximum of a list
$ for index in {1..5}; do
>     echo $((RANDOM/RANDOM)).${RANDOM}
> done
1.4122
2.15901
0.19098
0.6631
0.15050
$ for index in {1..5}; do
>     echo $((RANDOM/RANDOM)).${RANDOM}
> done | sort -n
0.1455
0.32373
0.9598
1.18896
1.2768
$ for index in {1..5}; do
>     echo $((RANDOM/RANDOM)).${RANDOM}
> done | sort -n | tail -n1
2.21350
```

Pipes: Exit codes

- The exit status of a pipeline is the exit status of the last command in the pipeline
- Bash provides an array variable `PIPESTATUS` containing a list of exit status values from the processes in the most-recently-executed foreground process

```
$ for index in {1..5}; do
>     echo $((RANDOM/RANDOM)).${RANDOM}
> done | sort -n | tail -n1 | grep 'x'
$ echo $?
1
$ for index in {1..5}; do
>     echo $((RANDOM/RANDOM)).${RANDOM}
> done | sort -n | tail -n1 | grep 'x'
$ echo ${PIPESTATUS[@]}
0 0 0 1
$ for index in {1..5}; do
>     echo $((RANDOM/RANDOM)).${RANDOM}
> done | sort -n | rm 2> /dev/null | tail -n1 | grep 'x'
$ echo ${PIPESTATUS[@]}
0 141 1 0 1 # 🔗Pipes must have a reader (rm is not) and a writer
```


If `pipefail` is enabled, the pipeline's return status is the value of the rightmost command to exit with a non-zero status, or zero if all commands exit successfully.

Process substitution

```
command <(list)    # No space between < and (
command >(list)    # No space between > and (
```

1. The process list is run asynchronously, and its input or output appears as a filename
2. This filename is passed as an argument to the current command as the result of the expansion

`>(command_list)`

writing to the file will provide input for `command_list` {Rarely needed!}

`<(command_list)`

the file passed as an argument should be read to obtain the output of `command_list`

The power of process substitution

Piping the stdout of a command into the stdin of another is a powerful technique. But, what if you need to pipe the stdout of multiple commands?

This is where process substitution comes in!

Process substitution

```
command <(list)   # No space between < and (
command >(list)   # No space between > and (
```

```
$ cat <(date)   # Equivalent to a pipeline
Wed 24 Jul 16:37:23 CEST 2019
$ echo <(date)
/dev/fd/63
# Here a pipeline is not enough!
$ cat <(date +'%T-%N') <(date +'%T-%N') <(date +'%T-%N')
16:43:16-957108564
16:43:16-958083851  # <(...) <(...) <(...) happen 'concurrently'*
16:43:16-957603584
$ echo <(date +'%T-%N') <(date +'%T-%N') <(date +'%T-%N')
/dev/fd/63 /dev/fd/62 /dev/fd/61
$ cat < <(date)
Wed 24 Jul 16:53:43 CEST 2019
$ echo < <(date)
# Why do you get a blank output here?
$ sdiff -s <(head Day_1.tex) <(head Day_2.tex)
\subtitle{Day 1}                          | \subtitle{Day 2}
\date{07.10.2019}                         | \date{08.10.2019}
```


* This is an incomplete statement, if taken as general rule: It is not the full story .

Ready to script!?

Yoda meditating in the Jökursárlón

Writing scripts

- Is it a Bash script? Take advantage of it!
- You learnt all flow constructs, keep them in mind
- Use meaningful, lower-case (or mixed-case) variable names!
- Does your script accept command line options? Implement a `-h` option!
- If environment variables affect the script behaviour, document it
- Do the error message get printed to the standard error?
- Take advantage of colours, if needed
- Do not make assumptions, be paranoic
- Use (associative) arrays for collections
- Avoid unnecessary pipes, use herestrings!
- Is the script getting large? Split it! { More tomorrow with functions }
- ...