Introduction to Bash scripting language

Day 3

Alessandro Sciarra

Z02 – Software Development Center

Organised by the CSC Frankfurt

28.10.2020

Topics of the day

1. File descriptors and redirection
2. Compound commands
3. Functions
4. Built in VS external programs
5. Script autocompletion
6. GNU Readline

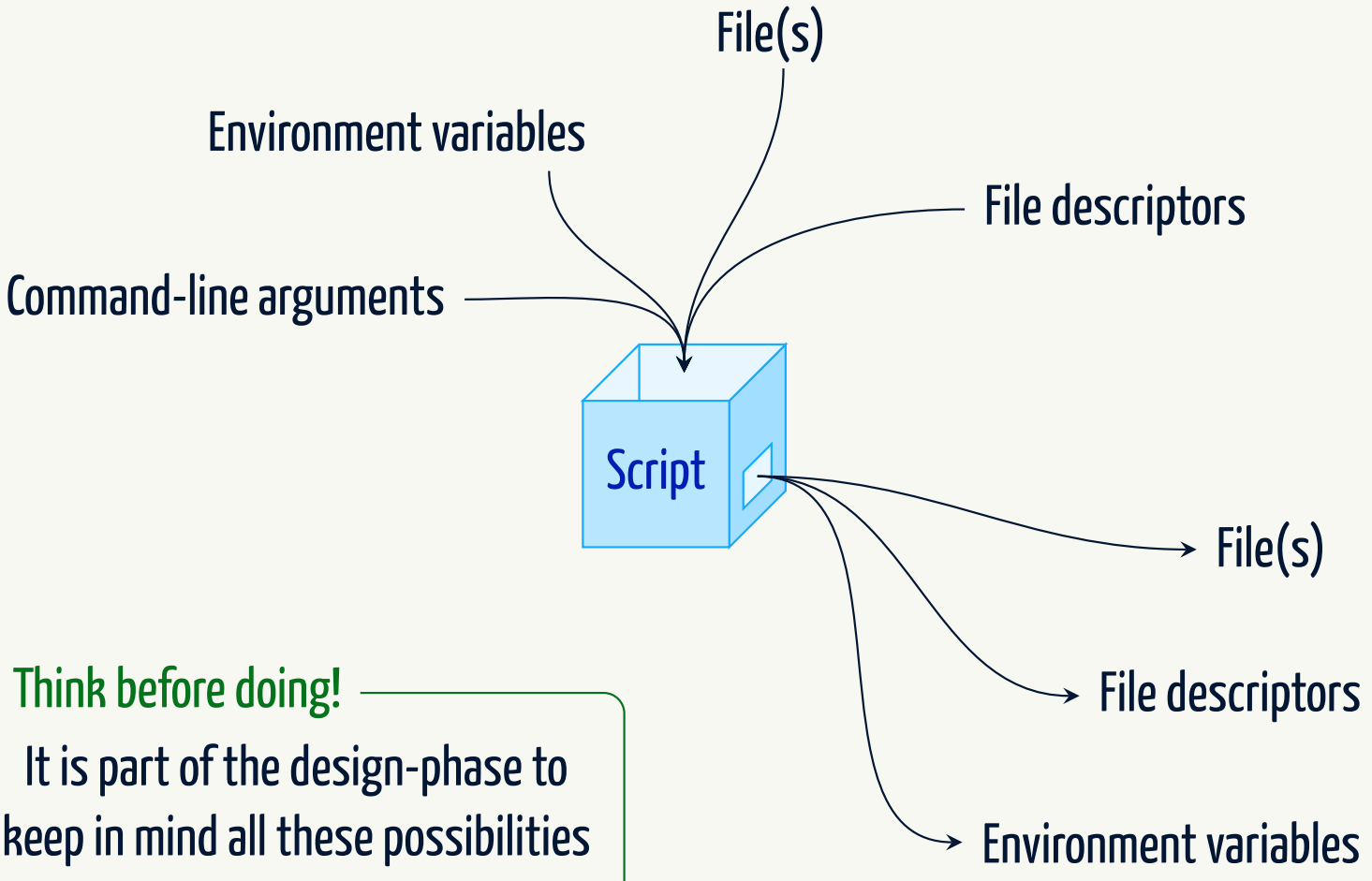
You already know almost everything

Today we will complete the picture about bash for almost all purposes, discovering some useful features for your daily work!

File descriptors and redirection

Horses

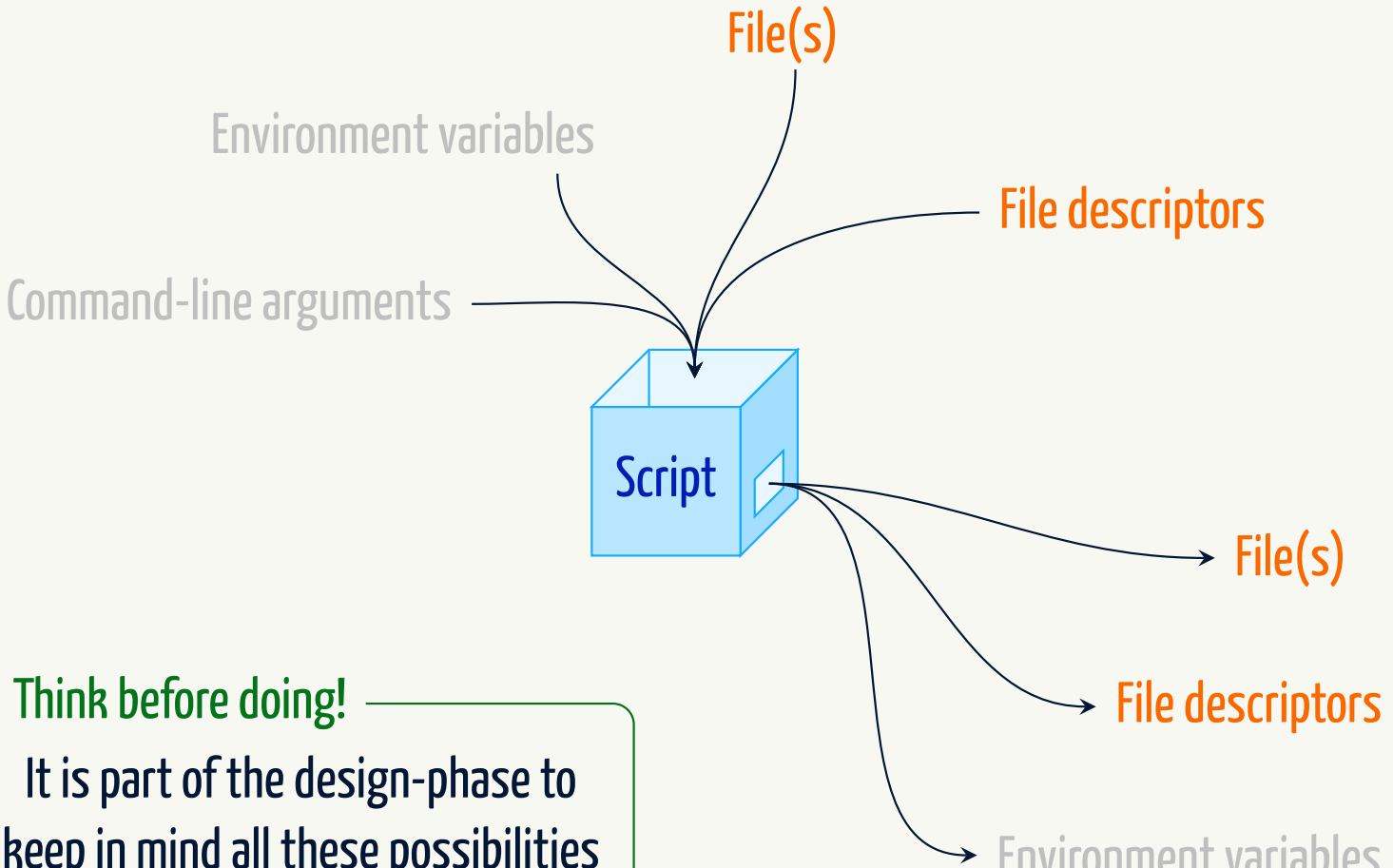
The Bash script flow



Think before doing!

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

The Bash script flow



Think before doing!

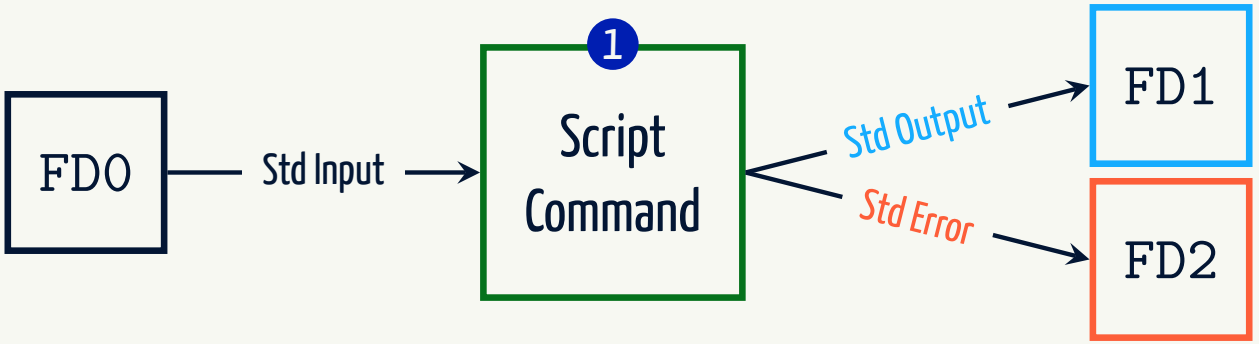
It is part of the design-phase to keep in mind all these possibilities and take advantage of them.

We discussed these in detail yesterday.

File descriptors: a graphical overview

By default

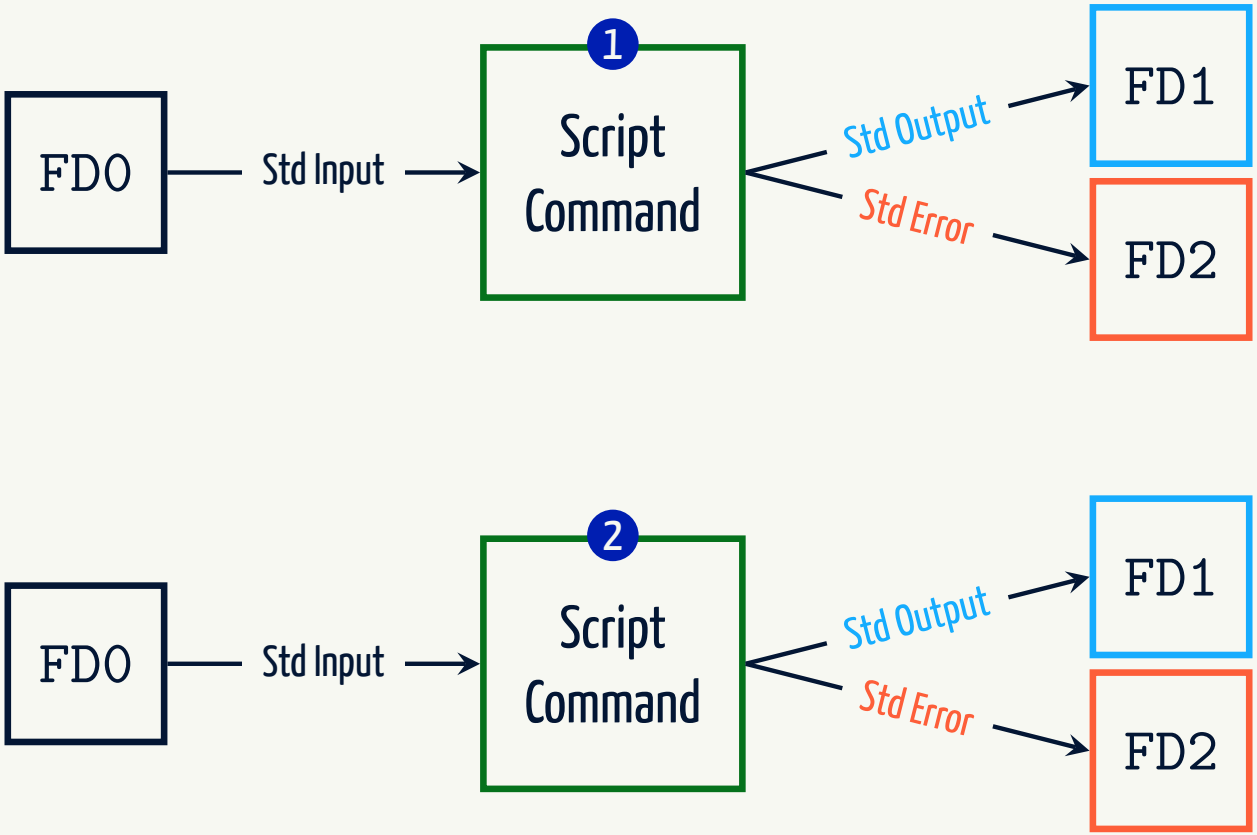
Every new process starts with **three** open file descriptors.



File descriptors: a graphical overview

By default

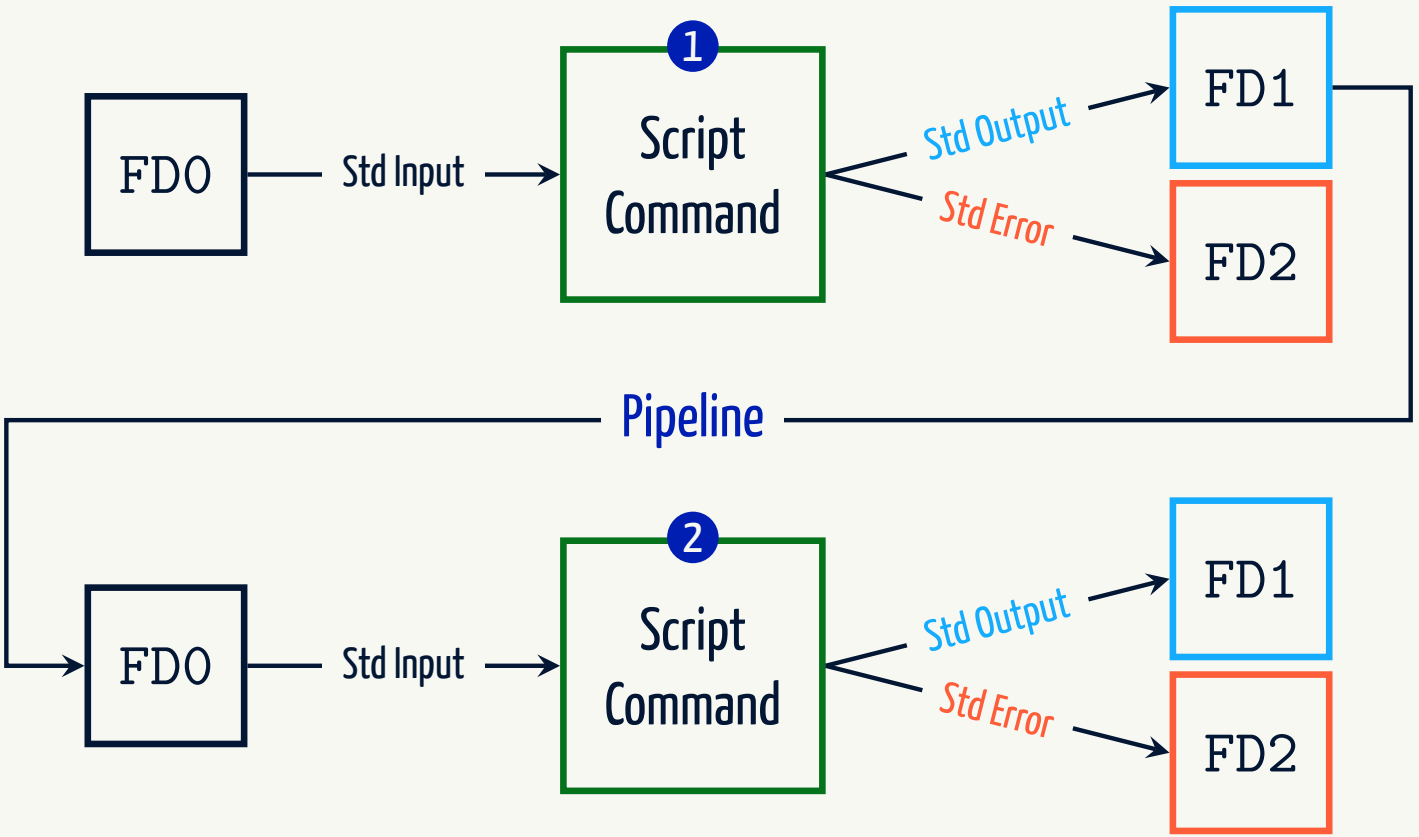
Every new process starts with **three** open file descriptors.



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

```
19 # Standard INPUT redirection
20 $ wc -l < Quote.txt
21 2
22 # Standard ERROR redirection
23 $ rm Quote.ttx
24 rm: cannot remove 'Quote.ttx': No such file or directory
25 $ rm Quote.ttx > Error.log
26 rm: cannot remove 'Quote.ttx': No such file or directory
27 $ ls; wc -l Error.log
28 Error.log   Quote.txt
29 0 Error.log
30 $ rm Quote.ttx 2> Error.log
31 $ wc -l Error.log
32 1 Error.log
33 $ rm Quote.ttx 2>> Error.log
34 $ cat Error.log
35 rm: cannot remove 'Quote.ttx': No such file or directory
36 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

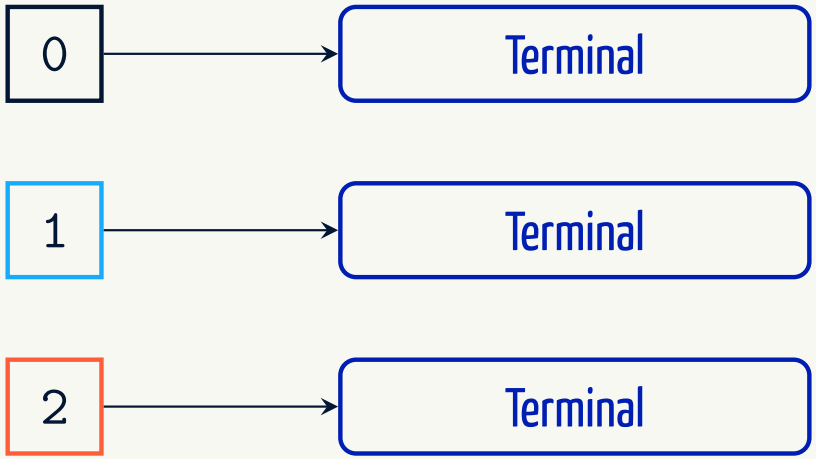
I/O FDs: `[n]<>word`

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

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

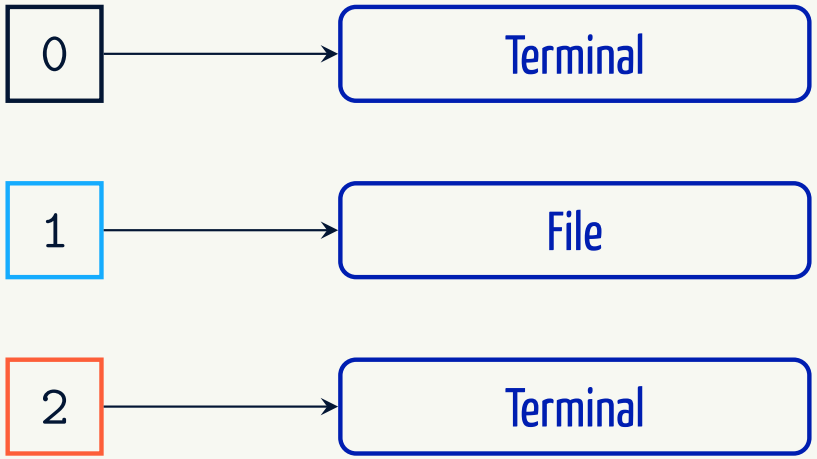
A few more basic examples

```
# Standard process situation
$ command
```



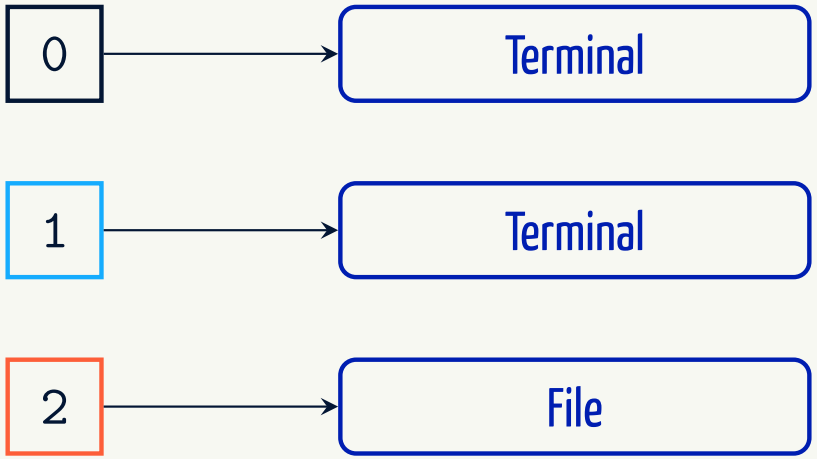
A few more basic examples

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



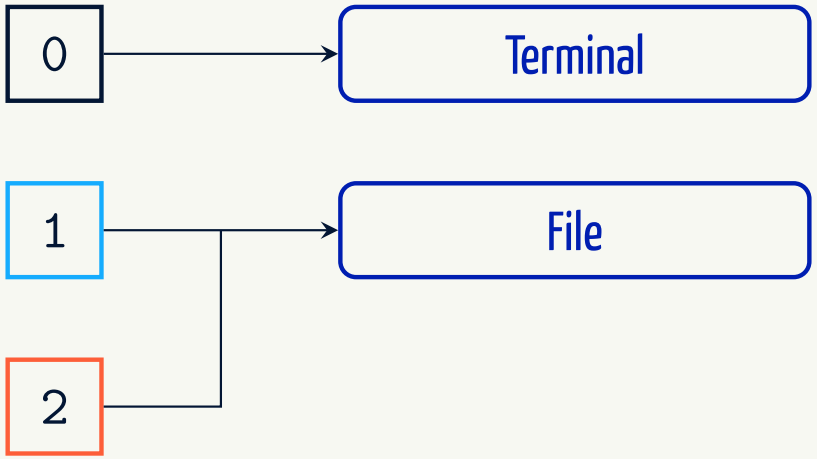
A few more basic examples

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



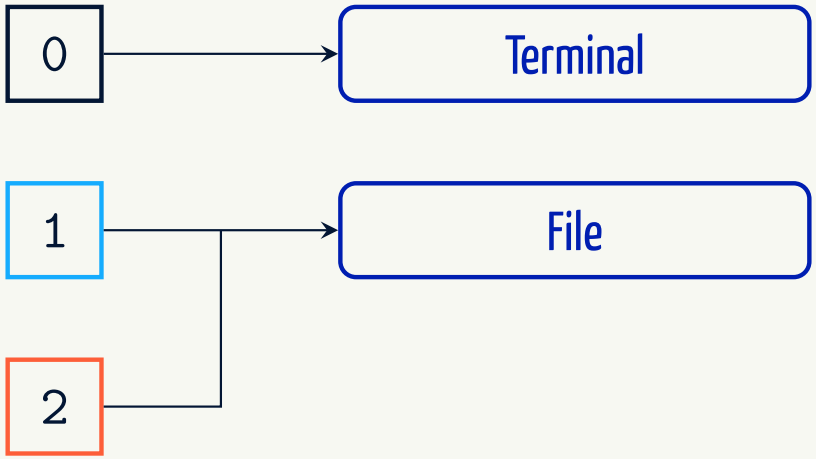
A few more basic examples

```
# 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
```



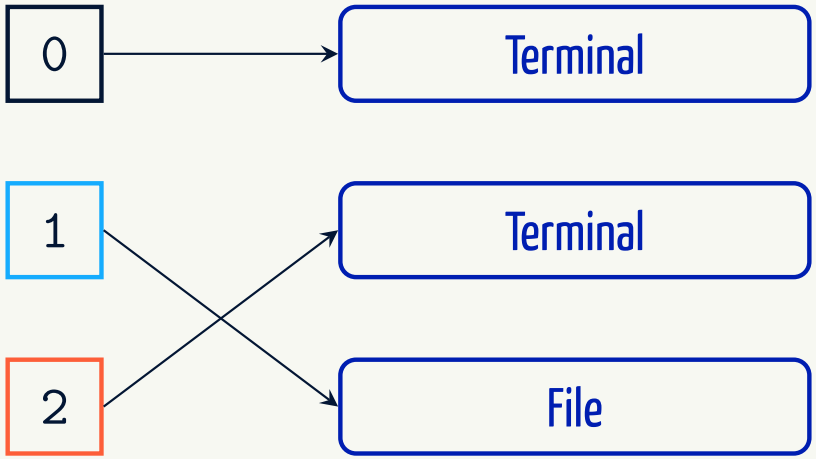
A few more basic examples

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



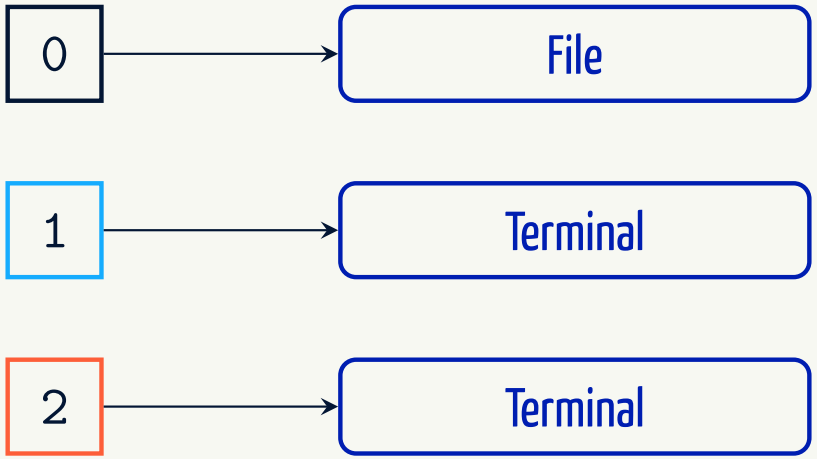
A few more basic examples

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



A 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"
```

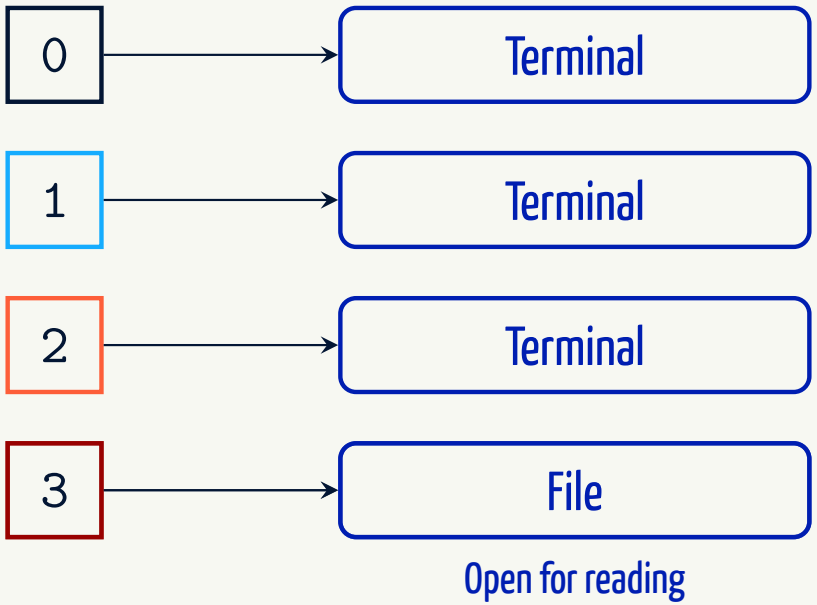
Otherwise, for local changes, command grouping helps:

```
# 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`

A few more advanced examples

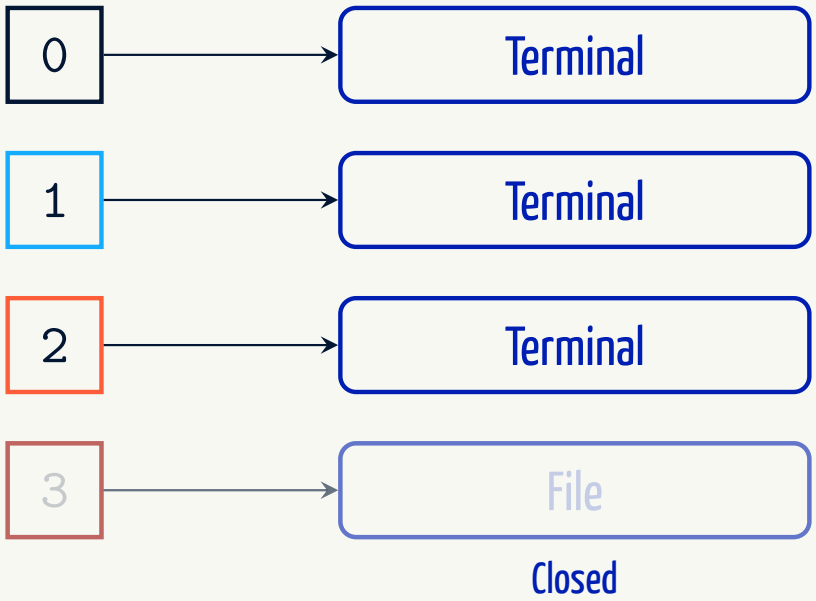
```
# Open a file for reading using a custom file descriptor
$ exec 3< File
# grep "foo" <&3 # NOTE: You can't rewind a fd in Bash!
# ...                    Close and open it again in case
```



File descriptor numbers range from 0 to 255.

A few more advanced examples

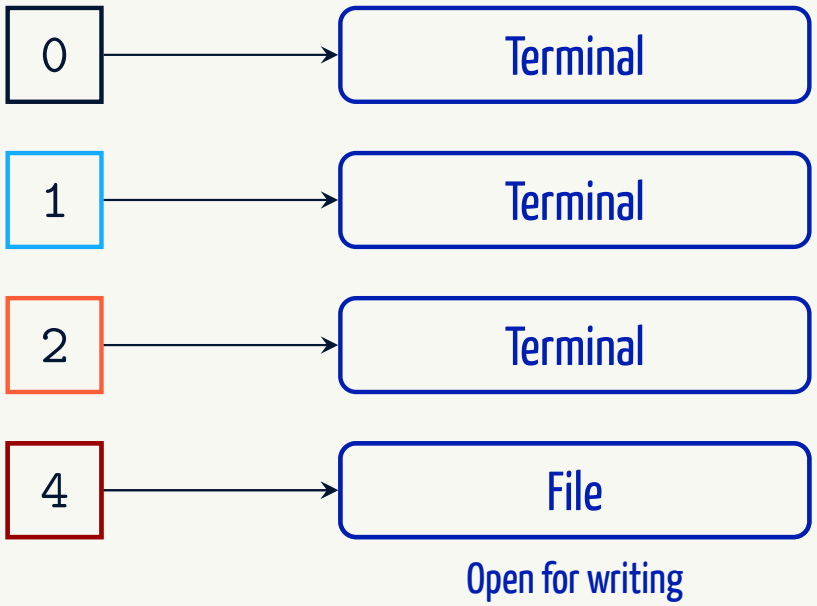
```
# Open a file for reading using a custom file descriptor
$ exec 3< File
# grep "foo" <&3 # NOTE: You can't rewind a fd in Bash!
# ...                    Close and open it again in case
$ exec 3<&-
```



File descriptor numbers range from 0 to 255.

A few more advanced examples

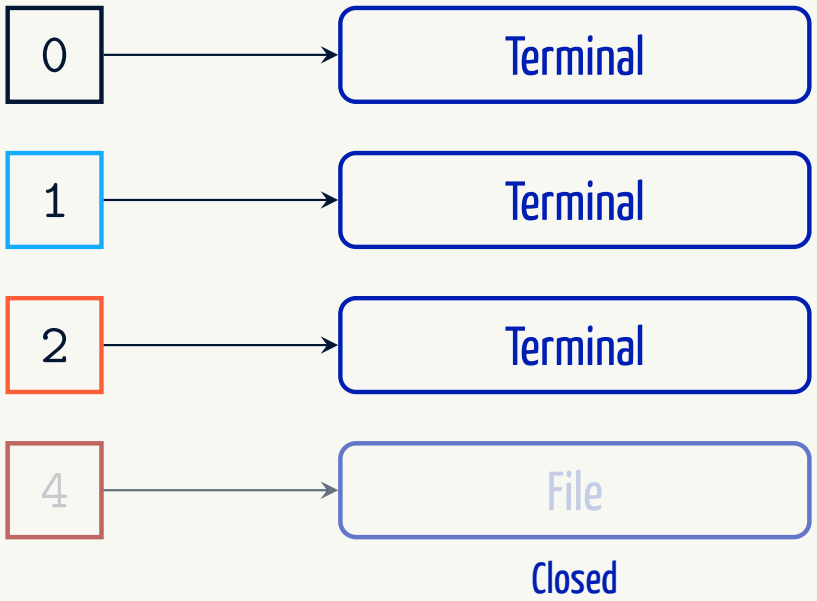
```
# Open a file for writing using a custom file descriptor
$ exec 4> File
# echo "foo" >&4
# ...
```



File descriptor numbers range from 0 to 255.

A few more advanced examples

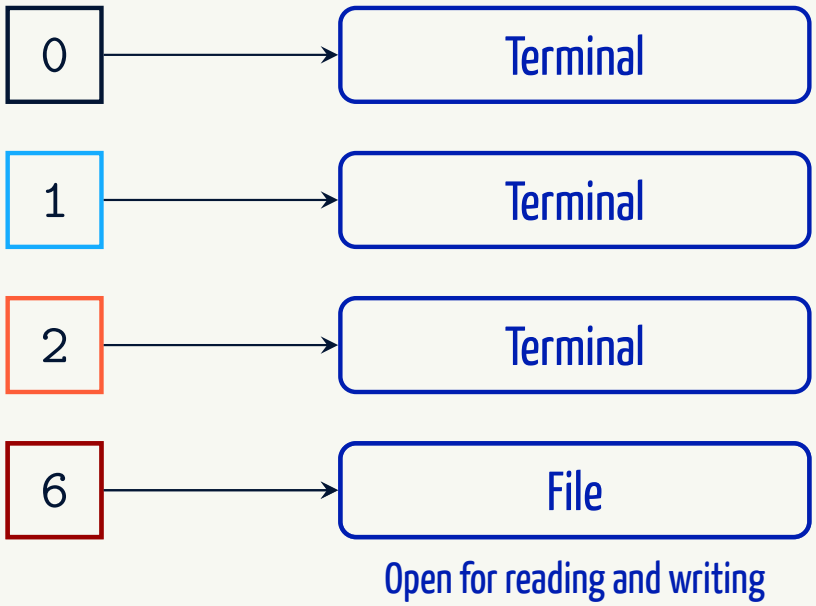
```
# Open a file for writing using a custom file descriptor
$ exec 4> File
# echo "foo" >&4
# ...
$ exec 4>&-
```



File descriptor numbers range from 0 to 255.

A few more advanced examples

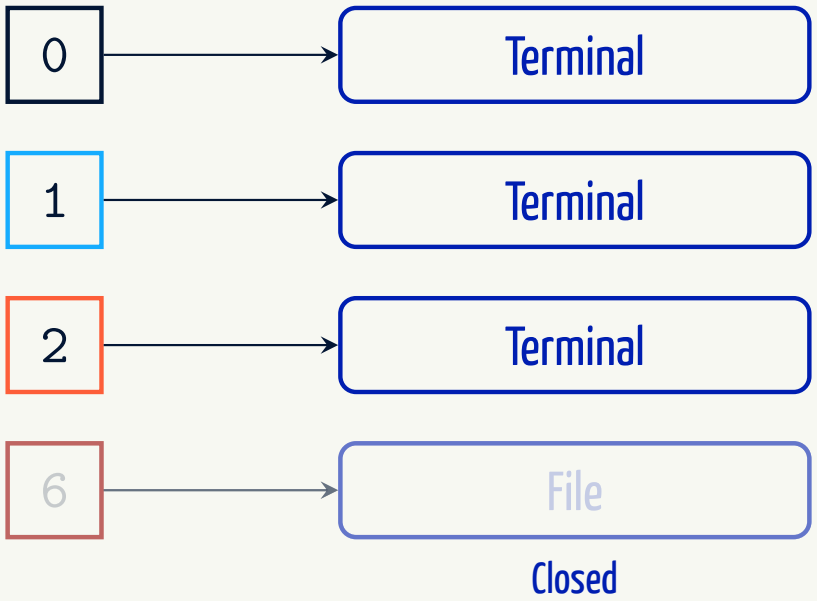
```
$ echo "foo bar" > File
# Open a file both for writing and reading with a custom FD
$ exec 6<> File
$ read -n 3 var <&6  # read the first 3 characters from FD 6
$ echo $var
$ echo -n + >&6      # write "+" at 4th position
```



File descriptor numbers range from 0 to 255.

A few more advanced examples

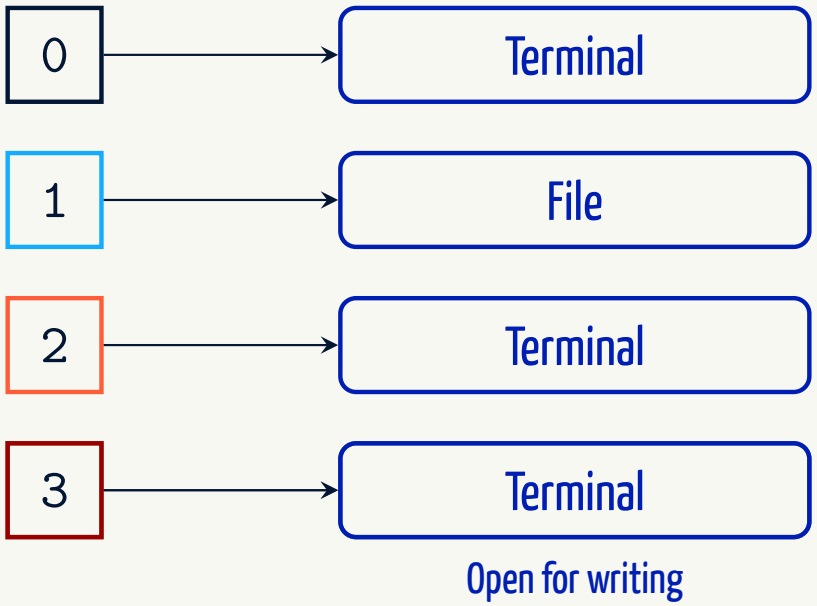
```
$ echo "foo bar" > File
# Open a file both for writing and reading with a custom FD
$ exec 6<> File
$ read -n 3 var <&6  # read the first 3 characters from FD 6
$ echo $var
$ echo -n + >&6      # write "+" at 4th position
$ exec 6>&-          # close FD 6
$ cat File
```



File descriptor numbers range from 0 to 255.

A few more advanced examples

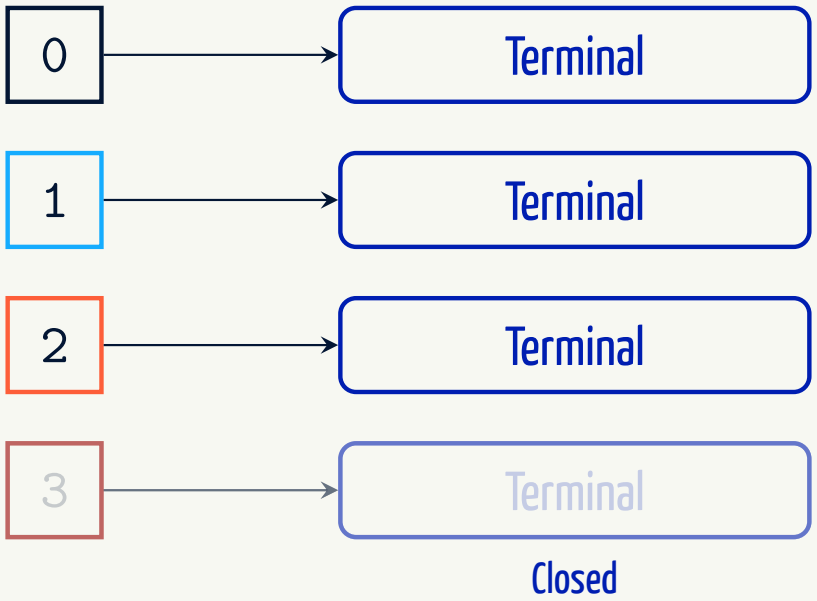
```
# Redirect standard output temporarily to file
$ exec 3>&1 1> File
$ date
$ echo "This goes to File without redirection!"
$ date
```



File descriptor numbers range from 0 to 255.

A 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.

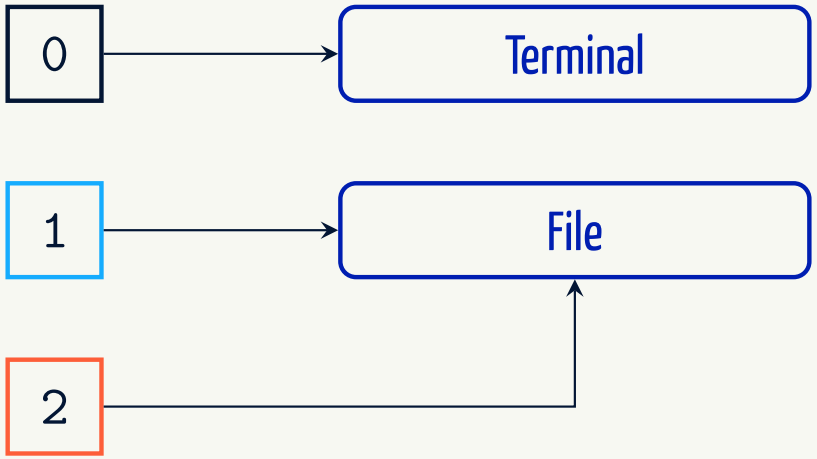
Redirecting standard output and error to the same place

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 }

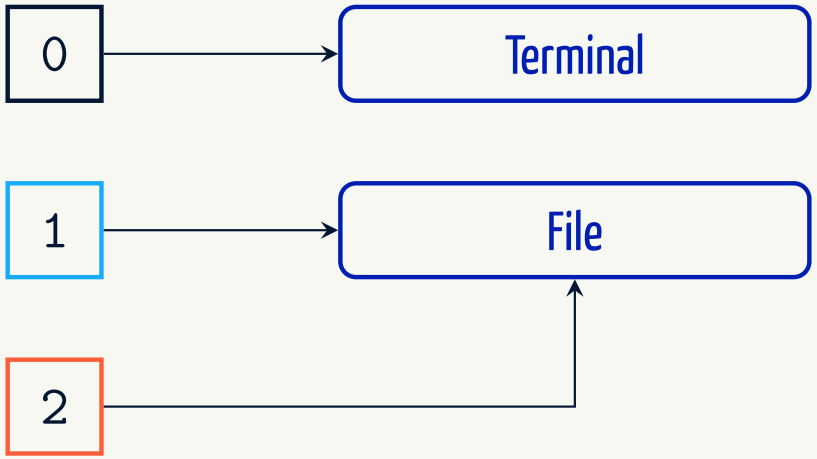


Redirecting standard output and error to the same place

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.
```



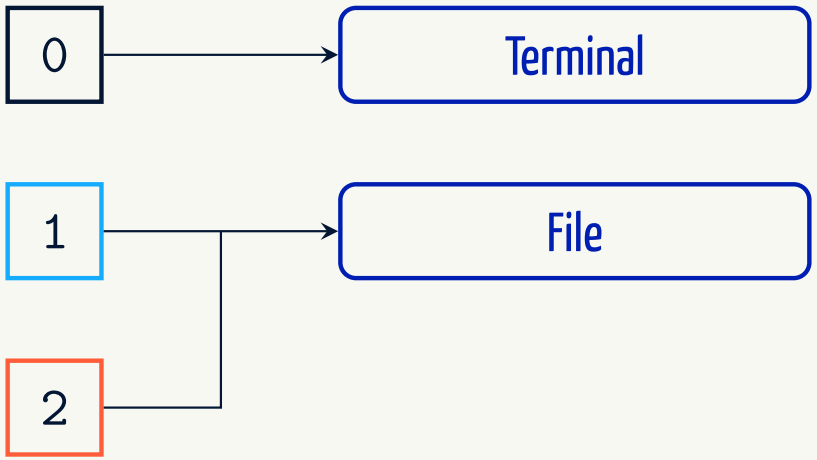
Redirecting standard output and error to the same place

Why does the naive way fail?

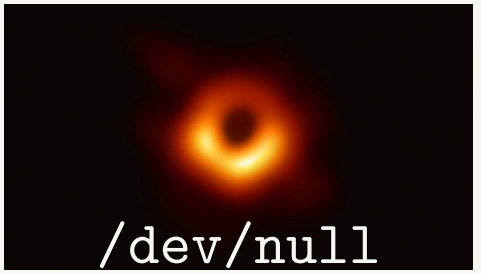
```
$ command > File 2> File    # WRONG CODE!
```

How should I fix this?

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



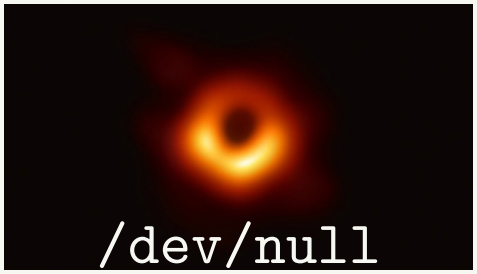
A kind of black hole in the operating system




```
$ 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 operating 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 is 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: Wednesday
# 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: Wednesday
# 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 Heredocs

  { 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 Heredocs

  { 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

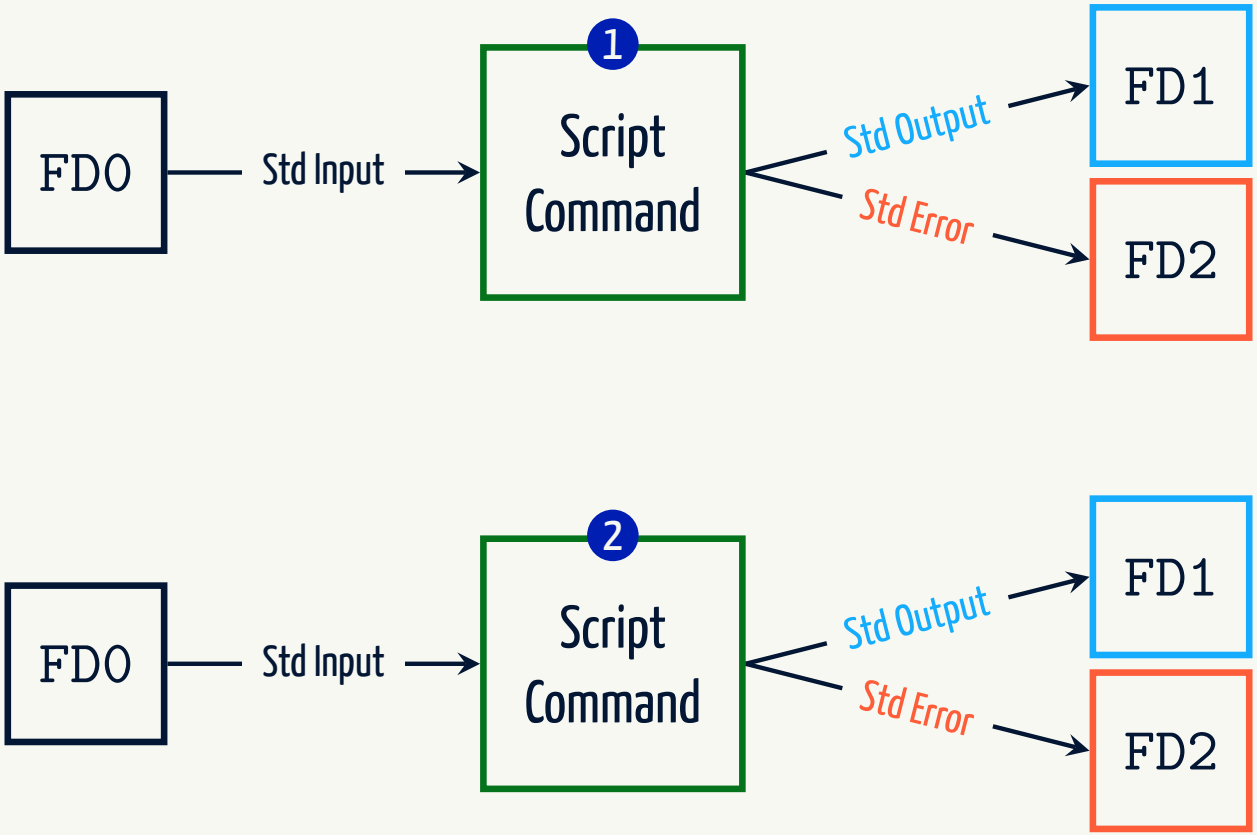
```
command [n]<<< STRING
```

```
$ 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

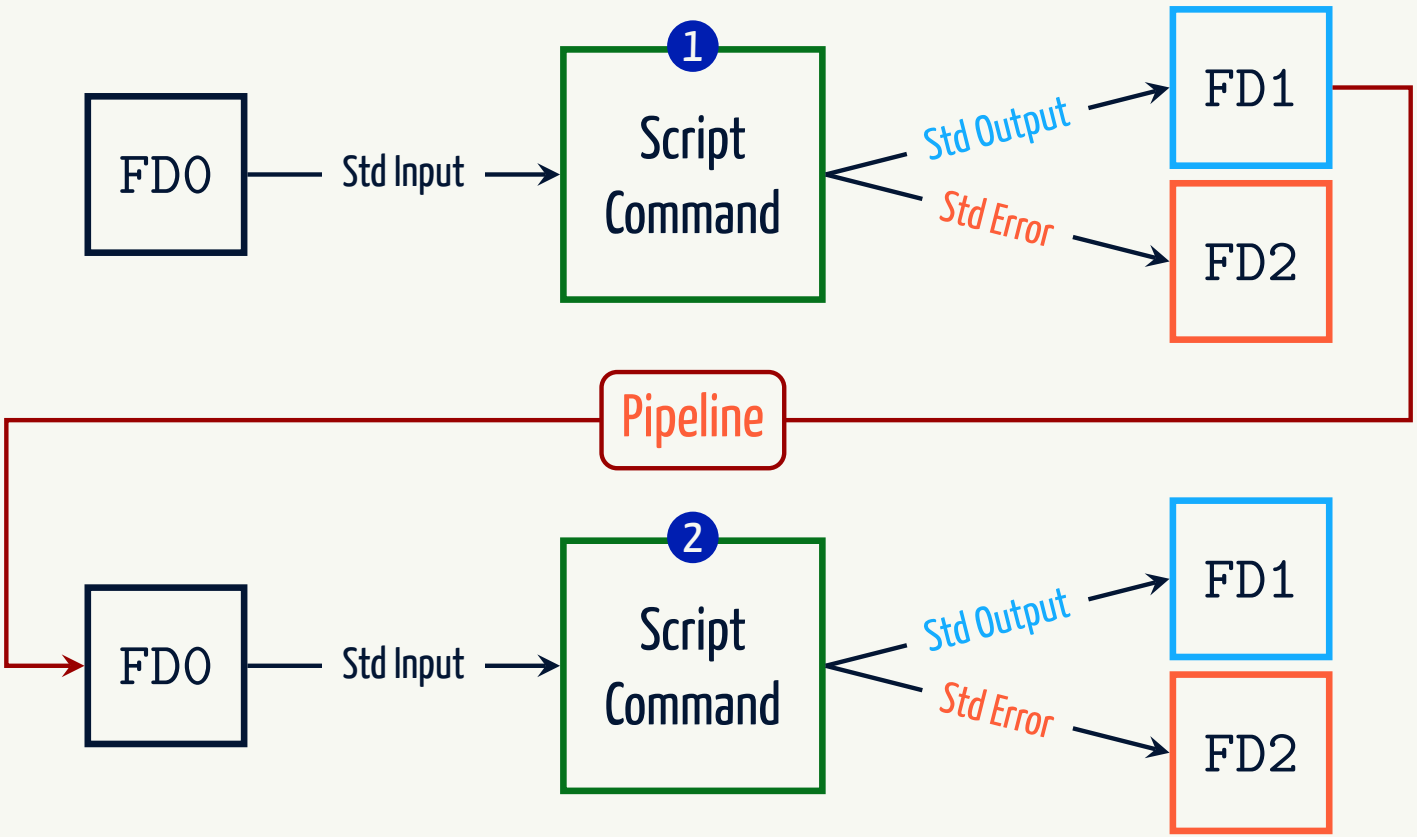
Every new process starts with **three** open file descriptors.



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{26.10.2019}                         | \date{27.10.2019}
```


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

Compound commands

The Skogafoss cascade

Compound commands

- `if` statements
- `for` loops
- `while`, `until` loops
- `[[` keyword
- `case`, `select` constructs

- Subshells
- Command grouping
- Arithmetic evaluation

  { Slightly different from the Arithmetic expansion we already discussed }

- [Functions] { Discussed in detail in a separate section }

Compound commands

- `if` statements
- `for` loops
- `while`, `until` loops
- `[[` keyword
- `case`, `select` constructs

} Already discussed in detail

- Subshells
- Command grouping
- Arithmetic evaluation

} What we are going to discuss

{ Slightly different from the Arithmetic expansion we already discussed }

- [Functions] { Discussed in detail in a separate section }

Strictly speaking not a compound command, but they work in a similar way

Subshells

Definition

A subshell is **a child process** but it is one **that inherits more than a normal external command** does. It can see all the variables of your script, not just the ones that have been exported to the environment.

Unix process

Every process on a Unix system has its own parcel of memory, for holding its variables, its file descriptors, its copy of the Environment inherited from its parent process, and so on. The changes to the variables (and other private information) in one process do not affect any other processes currently running on the system.

Use them consciously

Forking a subshell leads to a speed penalty which often is irrelevant but which you should keep in mind!

Subshells

Definition

A subshell is **a child process** but it is one **that inherits more than a normal external command** does. It can see all the variables of your script, not just the ones that have been exported to the environment.

- It is explicitly forced using the parenthesis `(...)`
- Changes e.g. to variables done in a subshell are not remembered when exiting the subshell: A subshell can be thought as a temporary shell!
- There are many instances in which a shell creates a subshell without parentheses being placed by the programmer
  - In pipelines ⟵ **Every command in a pipeline is run in its own subshell!**
  - In command substitution
  - Executing other programs or scripts
  - In any external command that executes new shells (e.g. `awk`, `sed`, `perl`)
  - In process substitution
  - In backgrounded commands and coprocs

From Bash v4.2 the shell option `lastpipe` can be enabled so that the last command in a pipeline is not run in a subshell.

Subshells: Examples

```
# Changes in a subshell do not propagate back
$ aVar="Hello"; pwd
/home/sciarra/Documents
$ ( aVar="Goodbye"; echo "${aVar}" ); echo "${aVar}"
Goodbye
Hello
$ ( cd /tmp; pwd ); pwd
/tmp
/home/sciarra/Documents
# It is often a feature to take advantage of!
$ (cd /tmp || exit 1; tar ...)
$ (source ~/AuxiliaryBashTools.bash; ...)
# In a subshell the script variable are accessible
$ ( echo "${aVar} from the subshell" )
Hello from the subshell
# Implicit subshells: be aware of them!
$ echo "Goodbye" | read aVar; echo "${aVar}"
Hello
$ read aVar <<< "Goodbye"; echo "${aVar}"; unset aVar
Goodbye
```


Advanced reading about pipelines and subshells: Why can't I pipe data to read?

Command grouping

We already said something about it, let us go through once again

- Commands may be grouped together using curly braces `{...}`
- Command groups allow a collection of commands to be considered as a whole with regards to redirection and control flow
- All compound commands such as `if` statements and `while` loops do this as well, but command groups do **only** this
- Command groups are executed in the same shell as everything else, NOT in a new one!

```
$ { echo "$(date)"
>   rsync -av . /backup; echo "$(date)"; } >backup.log 2>&1
```

Command grouping

We already said something about it, let us go through once again

- Commands may be grouped together using curly braces `{...}`
- Command groups allow a collection of commands to be considered as a whole with regards to redirection and control flow
- All compound commands such as `if` statements and `while` loops do this as well, but command groups do **only** this
- Command groups are executed in the same shell as everything else, NOT in a new one!

```
$ { echo "$(date)"
>   rsync -av . /backup; echo "$(date)"; } >backup.log 2>&1
```


Why is this semicolon absolutely **mandatory**?

Command grouping

We already said something about it, let us go through once again

- Commands may be grouped together using curly braces `{...}`
- Command groups allow a collection of commands to be considered as a whole with regards to redirection and control flow
- All compound commands such as `if` statements and `while` loops do this as well, but command groups do **only** this
- Command groups are executed in the same shell as everything else, NOT in a new one!

```
$ { echo "$(date)"
>   rsync -av . /backup; echo "$(date)"; } >backup.log 2>&1
```


Why is this semicolon absolutely **mandatory**?

Because otherwise the closing curly bracket would be a argument of the final command in the group, `echo` in this case

Command grouping

We already said something about it, let us go through once again

- Commands may be grouped together using curly braces `{...}`
- Command groups allow a collection of commands to be considered as a whole with regards to redirection and control flow
- All compound commands such as `if` statements and `while` loops do this as well, but command groups do **only** this
- Command groups are executed in the same shell as everything else, NOT in a new one!

```
$ { echo "$(date)"
>   rsync -av . /backup; echo "$(date)"; } >backup.log 2>&1
```

The above example truncates and opens the file backup.log on stdout, then points stderr at where stdout is currently pointing (backup.log), then runs each command with those redirections applied. The file descriptors remain open until all commands within the command group complete before they are automatically closed. This means backup.log is only opened a single time, not opened and closed for each command.

Command grouping

We already said something about it, let us go through once again

```
$ TIMEFORMAT='%3R seconds'
$ time for((i=0; i<10000; i++)); do
>   echo "Hello ${i}" >> FileInFor.dat
> done
11.451 seconds
$ time for((i=0; i<10000; i++)); do
>   echo "Hello ${i}"
> done >> FileOutFor.dat
0.054 seconds
# The above for-loops do the same thing
$ diff FileInFor.dat FileOutFor.dat; echo $?
0
```

The above example truncates and opens the file backup.log on stdout, then points stderr at where stdout is currently pointing (backup.log), then runs each command with those redirections applied. The file descriptors remain open until all commands within the command group complete before they are automatically closed. This means backup.log is only opened a single time, not opened and closed for each command.

Arithmetic evaluation: The `let` builtin

- Sometimes we want to do arithmetic instead of string operations
- One way to do so is to use the `let` builtin

```
$ aVar=4+5; echo "${aVar}"
4+5
$ let aVar=4+5; echo "${aVar}"; unset aVar
9
```

- However, it requires quotes to use arithmetic operators {help let}

```
$ let aVar=2<3
bash: 3: No such file or directory
$ let aVar='2<3'; echo "${aVar}"; unset aVar
1
```

- `let` accepts more than one expression

```
$ let aVar='2<3' bVar=3*7; echo "${aVar} ${bVar}"
1 21
$ unset aVar bVar
```

- If the last expression evaluates to 0, `let` returns 1; `let` returns 0 otherwise.

Arithmetic evaluation: The command grouping `( (...) )`

- `(( expression ))` is equivalent to `let "expression"`
- No quote is needed in it, since only arithmetic operations are there performed
- However, only one expression can be evaluated (not bad for the exit code)

```
$ (( aVar=7*3**2 )); echo "${aVar}"
63
$ (( aVar=1+${aVar}/20 )); echo "${aVar}"; unset aVar
4          # '${    }' not really needed*
```

- Although not a compound command, the arithmetic substitution uses the same syntax

```
$ (( aVar=7*3**2 )); echo "${aVar}"
63
$ echo "aVar=$(( 7*3**2 ))"; unset aVar
63
```

- Assignments in arithmetic substitution work but are confusing and should be avoided!

```
$ echo "_${bVar}_ $(( bVar=7*3**2 )) _${bVar}_"
__ 63 _63_
```


* Using the `${}` makes Bash use `''` for uninitialised variables and might trigger errors (without `${}` syntax, 0 is used for uninitialised referenced variables).

Arithmetic evaluation: The command grouping ((...))

- Arithmetic evaluation is very helpful in combination with conditionals

```
$ ((aVar=(5+2)*3))
$ if ((aVar == 21)); then
>   echo 'Blackjack!'
> fi
Blackjack!
```

- Because the inside of ((...)) is C-like, a variable (or expression) that evaluates to zero will be considered false for the purposes of the arithmetic evaluation. Then, because the evaluation is false, it will exit with a status of 1. Likewise, if the expression inside ((...)) is non-zero, it will be considered true; and since the evaluation is true, it will exit with status 0.

```
$ flag=0  # no error
$ while read line; do
>   if [[ ${line} == *err* ]]; then flag=1; fi
> done < inputfile
$ if ((flag)); then echo 'Houston, we have a problem!'; fi
```

Functions

The Keifarvatn lake

Functions, finally! Overview

- Functions are the last basic Bash feature we'll learn
  - They give you an incredible opportunity to structure and increase readability of your script
  - They can be used to split large script across multiple files in an elegant way
  - Learn them well, for real!
- Functions are a tricky world!
- They have several features that we might call **issues** if compared to other languages
- Indeed, functions in Bash are not as powerful as we might expect
  - → Return value
  - → Reusability
  - → Scope
  - → I/O + ...

> Don't bite the newbie for not understanding all this. Shell functions are totally f***ed.
>
> — Greg's Wiki

Do not be scared!

Learn, understand and use functions for what they are, not for what you would like them to be!

Basic syntax

```
# POSIX compliant syntax
NAME () COMPOUND-COMMAND [ REDIRECTIONS ]
# Totally equivalent syntax (in Bash), but not POSIX
function NAME [()] COMPOUND-COMMAND [ REDIRECTIONS ]
```

NAME:

A **sane** function name should be an alphanumeric string, maybe containing underscore and not starting with a number. However, insane names are accepted in Bash and, in principle, names like `:` or `[]{` are allowed. But please, avoid them! Really!! { Exploring allowed names }

COMPOUND-COMMAND:

The body of a function can be any compound command.

- `{ list; }` ⟵ Use this if you do not have a reason to use a different one
- `(list)` or `((expression))` or `[[ expression ]]`

REDIRECTIONS:

They take place when the function is called and they refer to the whole compound-command.

Functions: The most basic example

Just accomplish a **detached** task

Whenever a block of code can be executed as standalone, without needing either input information nor variables, it is straightforward to include it in a dedicated function

```
1  #!/bin/bash
2  function CreateListOfFiles()
3  {
4      printf "#%19s %20s %15s %20s %20s %6s      %s\n" \
5             "user" "group" "permissions"                \
6             "size(KB)" "permissions" "type" "path"
7      find "${PWD}" -printf "%20u %20g %15m %20k %20M %6y      %p\n"
8  }
```

Note

This will do absolutely nothing when run. This is because it has only been stored in memory, much like a variable, but it has not yet been called.

Functions: The most basic example

Just accomplish a **detached** task

Whenever a block of code can be executed as standalone, without needing either input information nor variables, it is straightforward to include it in a dedicated function

```
1  #!/bin/bash
2  function CreateListOfFiles()
3  {
4      printf "#%19s %20s %15s %20s %20s %6s      %s\n" \
5             "user" "group" "permissions"                \
6             "size(KB)" "permissions" "type" "path"
7      find "${PWD}" -printf "%20u %20g %15m %20k %20M %6y      %p\n"
8  }
```

Note

This will do absolutely nothing when run. This is because it has only been stored in memory, much like a variable, but it has not yet been called.

```
10 CreateListOfFiles
```

Variables in functions and their scopes

- Variable in Bash are by definition **global**!
- Variables declared using the `local` builtin have a lifetime limited to the function scope
- Bash uses **dynamic scoping** to control a variable's visibility within functions
  - In a function all variables visible in the caller are visible and might be changed
  - Declaring a local variable with the same name of an already existing variable shadows the variable from the caller, whose value cannot be retrieved from within the function.

```
1  #!/bin/bash
2  LevelOne() {
3      LevelTwo
4      echo "In ${FUNCNAME}, aVar = ${aVar}"
5  }
6  LevelTwo() {
7      local aVar; aVar="${FUNCNAME} local"
8      LevelThree
9  }
10 LevelThree() { echo "In ${FUNCNAME}, aVar = ${aVar}"; }
11 aVar='global'; LevelOne
```

Variables in functions and their scopes

- Variable in Bash are by definition **global**!
- Variables declared using the `local` builtin have a lifetime limited to the function scope
- Bash uses **dynamic scoping** to control a variable's visibility within functions
  - In a function all variables visible in the caller are visible and might be changed
  - Declaring a local variable with the same name of an already existing variable shadows the variable from the caller, whose value cannot be retrieved from within the function.

```
$ ./script.bash
In LevelThree, aVar = LevelTwo local
In LevelOne, aVar = global
```

Variables in functions and their scopes

- Variable in Bash are by definition **global**!
- Variables declared using the `local` builtin have a lifetime limited to the function scope
- Bash uses **dynamic scoping** to control a variable's visibility within functions
  - In a function all variables visible in the caller are visible and might be changed
  - Declaring a local variable with the same name of an already existing variable shadows the variable from the caller, whose value cannot be retrieved from within the function.
- Unless you have a reason not to do so, declare all variables in functions as `local`
- Do not assign a value to a local variable at declaration, because you might obfuscate/loose an exit code!

```
$ function Test { aVar="$(exit 1)"; echo $?; }; Test
1  # Fine, but we pollute caller with 'aVar' variable
$ function Test { local aVar="$(exit 1)"; echo $?; }; Test
0  # <- local's exit code!
$ function Test { local aVar; aVar="$(exit 1)"; echo $?; }; Test
1  # GOOD CODE
```

Variables in functions and their scopes

- Variable in Bash are by definition **global**!
- Variables declared using the `local` builtin have a lifetime limited to the function scope
- Bash uses **dynamic scoping** to control a variable's visibility within functions
  - In a function all variables visible in the caller are visible and might be changed
  - Declaring a local variable with the same name of an already existing variable shadows the variable from the caller, whose value cannot be retrieved from within the function.
- Unless you have a reason not to do so, declare all variables in functions as `local`
- Do not assign a value to a local variable at declaration, because you might obfuscate/loose an exit code!
- If a variable at the current local scope is unset, it will remain so until it is reset in that scope or until the function returns. Once the function returns, any instance of the variable at a previous scope will become visible.
  However, if the unset acts on a variable at a previous scope, any instance of a variable with that name that had been shadowed will become visible!

Passing arguments to functions

So far so good

Delegating tasks to functions, even if the task requires variables, is quite straightforward, provided one keeps scope rules in mind

Passing arguments to functions

So far so good

Delegating tasks to functions, even if the task requires variables, is quite straightforward, provided one keeps scope rules in mind

The Pandora's box

> When a function is executed, the arguments to the function become the positional parameters during its execution. The special parameter # that expands to the number of positional parameters is updated to reflect the change. Special parameter 0 is unchanged.
>
> Bash manual

Passing arguments to functions

So far so good

Delegating tasks to functions, even if the task requires variables, is quite straightforward, provided one keeps scope rules in mind

The Pandora's box

> When a function is executed, the arguments to the function become the positional parameters during its execution. The special parameter # that expands to the number of positional parameters is updated to reflect the change. Special parameter 0 is unchanged.
>
> Bash manual

- Arguments to functions are meant to provide **input** to a function
- Bash strictly uses call-by-value semantics
- You can't pass arguments "by reference"

  { at least not until Bash 4.3 (and even there the `declare -n` mechanism has serious security flaws) }
- Passing the name of a variable to a function, which should use it, requires `eval` acrobatics and it should be as far as possible avoided!

Passing arguments to functions: Example

```
1  #!/bin/bash
2  function SecondsToTimeStringWithDays()
3  {
4      local inputTime days hours minutes seconds
5      inputTime=$1
6      ((    days=(inputTime)/86400 ))
7      ((   hours=(inputTime - days*86400)/3600 ))
8      (( minutes=(inputTime - days*86400 - hours*3600)/60 ))
9      (( seconds=inputTime%60 ))
10     printf "%d-%02d:%02d:%02d\n" \
11             "${days}" "${hours}" "${minutes}" "${seconds}"
12 }
13 for example in 31536000 86400 3600; do
14     SecondsToTimeStringWithDays ${example}
15 done
```

```
18 $ ./scriptAbove
19 365-00:00:00
20 1-00:00:00      # What is bad in the function above?
21 0-01:00:00
```

Passing arguments to functions: Example

```
22 #!/bin/bash

23 function SecondsToTimeStringWithDays() # Better code!
24 {
25     if [[ ! ${1} =~ ^[1-9][0-9]*$ ]]; then
26         echo "ERROR: Function ${FUNCNAME} wrongly called!"
27         return 1
28     fi
29     local inputTime days hours minutes seconds
30     inputTime=$1
31     ((    days=(inputTime)/86400 ))
32     ((   hours=(inputTime - days*86400)/3600 ))
33     (( minutes=(inputTime - days*86400 - hours*3600)/60 ))
34     (( seconds=inputTime%60 ))
35     printf "%d-%02d:%02d:%02d\n" \
36             "${days}" "${hours}" "${minutes}" "${seconds}"
37 }

38 for example in 31536000 86400 3600 ''; do
39     SecondsToTimeStringWithDays ${example}
40 done
```

The `return` builtin and functions' exit code

Functions' exit code

When executed, the exit status of a function is the exit status of the **last command executed** in the body.

```
$ help return
return: return [n]              # ATTENTION: 0 <= N <=255
Return from a shell function.

Causes a function or sourced script to exit with the
return value specified by N. If N is omitted, the
return status is that of the last command executed
within the function or script.

Exit Status:
Returns N, or failure if the shell is not executing
a function or script.
```

Do not use `return` to retrieve the function result!

Use `return` to early terminate a function and/or to give back an exit code!

How do I return something from a function?

How do I return something from a function?

It's simple: You don't!

How do I return something from a function?

It's simple: You don't!

1 Capturing standard output

- ✗ the function is executed in a subshell, i.e. variable assignments not seen in the caller
- ✗ **everything** printed by the function is captured (debug output!? → use stderr)
- ✓ good solution if performance is not critical

```
ExampleFunction() {
    echo "running foo()..."  >&2
    echo "this is my data"
}
aVar=$(ExampleFunction)
echo "ExampleFunction returned '${aVar}'"
```

How do I return something from a function?

It's simple: You don't!

1. Capturing standard output
2. Global variables
   - ✓ the function is **not** executed in a subshell → potentially much faster
   - ✗ if the function is executed in a subshell, the method fails!
   - ✗ the function cannot be used in a pipeline (as in any implicit subshell)

```
ExampleFunction() {
    globalVar="this is my data"
}
ExampleFunction
echo "ExampleFunction returned '${globalVar}'"
```

How do I return something from a function?

It's simple: You don't!

1. Capturing standard output
2. Global variables
3. Writing to a file
   - ✗ you need to manage a temporary file, which is always inconvenient
   - ✗ there must be a writable directory somewhere and sufficient space to hold the data therein
   - ✓ it will work regardless of whether your function is executed in a subshell

```
ExampleFunction() {
    echo "this is my data" > "$1"
}
# This is NOT solid code to handle temp files! Solid way
tmpfile=$(mktemp)   # GNU/Linux
ExampleFunction "${tmpfile}"
echo "ExampleFunction returned '$(cat < "${tmpfile}" )'"
rm "${tmpfile}"
```


If this were a real program, there would have been error checking, and a trap.

How do I return something from a function?

It's simple: You don't!

1. Capturing standard output
2. Global variables
3. Writing to a file
4. Dynamically scoped variables
   - ✗ if the function is executed in a subshell, the method fails!
   - ✗ this technique doesn't work with recursive functions
   - ✓ global variable namespace isn't polluted by the function return variable

```
ExampleFunction_implementation() {
    notGlobalVar="this is my data"
}
ExampleFunction() {
    local notGlobalVar; ExampleFunction_implementation
    # Do here something with 'notGlobalVar'
    echo "In ExampleFunction, got '${notGlobalVar}'"
}
# Here at the global scope, 'notGlobalVar' is not visible
ExampleFunction
echo "ExampleFunction returned '${notGlobalVar}'"
```

How do I return something from a function?

It's simple: You don't!

1 Capturing standard output

- ✗ the function is executed in a subshell, i.e. variable assignments not seen in the caller
- ✗ **everything** printed by the function is captured (debug output!? ⟶ use stderr)
- ✓ good solution if performance is not critical

2 Global variables

- ✓ the function is **not** executed in a subshell ⟶ potentially much faster
- ✗ if the function is executed in a subshell, the method fails!
- ✗ the function cannot be used in a pipeline (as in any implicit subshell)

3 Writing to a file

- ✗ you need to manage a temporary file, which is always inconvenient
- ✗ there must be a writable directory somewhere and sufficient space to hold the data therein
- ✓ it will work regardless of whether your function is executed in a subshell

4 Dynamically scoped variables

- ✗ if the function is executed in a subshell, the method fails!
- ✗ this technique doesn't work with recursive functions
- ✓ global variable namespace isn't polluted by the function return variable

Choose with care

Use the approach you prefer depending on the situation!

Splitting large scripts across multiple files

A Bash script should not be **too** large, though

Declaring a function does not mean to run it

You can collect functions in a separate file, which is meant to be sourced at need!

```
# Collection of tools

function ExtractColumnFromFile()
{
    # ...
}

function CalculateSizeOfFiles()
{
    # ...
}

function ReportOnLargestDirectories()
{
    # ...
}
```

Splitting large scripts across multiple files

A Bash script should not be **too** large, though

```
#!/bin/bash

source /home/sciarra/Script/UtilityFunctions.bash
source /home/sciarra/Script/UtilityFunctions_nice.bash
source /home/sciarra/Script/UtilityFunctions_cool.bash

# Call to functions (only, maybe)
```

- As long as the sourced files contain functions only,
  - the **sourcing order does not matter** and
  - functions in a file can even use functions in another file!
- The shebang can/should be included in the main file only!
- Prevent auxiliary files from being run, only sourced!

```
if [[ "${BASH_SOURCE[0]}" == "${0}" ]]; then
  echo "Script \"${BASH_SOURCE[0]}\" can only been sourced!" 2>&1
  exit -1
fi
```

Functions: Miscellaneous

- Functions can be marked as read-only using the `readonly` builtin
- Functions can be unset via `unset -f`
- Functions can be recursive

```
1 function CountTill5From()
2 {
3     if [[ $1 < 5 ]]; then
4         echo $1; CountTill5From $(($1 + 1))
5     fi
6 }
```

- The `FUNCNEST` variable may be used to limit the depth of the function call stack and restrict the number of function invocations { By default, no limit is placed on the number of recursive calls }

```
$ FUNCNEST=2; CountTill5From()
1
2
bash: CountTill5From: maximum function nesting level exceeded (2)
$ unset -v FUNCNEST
```

Functions: Miscellaneous

About recursion

- «To iterate is human, to recur, divine.» – **L. Peter Deutsch**
- There are cases where it is needed
- In Bash rarely, though
- The fact that by default in Bash there is no limit to the number of function invocations is something to have clear in mind!
- For example, what does the following code do? **DON'T RUN IT!**

```
:(){ :|:& };:    # Do NOT run this line!
```

Functions: Miscellaneous

About recursion

- «To iterate is human, to recur, divine.» – **L. Peter Deutsch**
- There are cases where it is needed
- In Bash rarely, though
- The fact that by default in Bash there is no limit to the number of function invocations is something to have clear in mind!
- For example, what does the following code do? **DON'T RUN IT!**

```
:(){ :|:& };:    # Do NOT run this line!
```

It is equivalent to

```
# Do NOT run this code!
function bomb() { bomb | bomb & }; bomb
```

A **fork bomb** is a denial-of-service attack wherein a process continually replicates itself to deplete available system resources, **slowing down or crashing the system** due to resource starvation.

Wikipedia

A last big warning: The `eval` builtin

```
$ help eval
eval: eval [arg ...]
Execute arguments as a shell command.

Combine ARGs into a single string, use the result as input
to the shell, and execute the resulting commands.

Exit Status:
Returns exit status of command or success if command is null.
```

- «`eval` is a common misspelling of **evil**» – Greg's Wiki
- It causes your code to be parsed twice instead of once; this means that, for example, if your code has variable references in it, the shell's parser will evaluate the contents of that variable. This can lead to unexpected results!

Advanced, but very informative article which you should spend some time at some point on.

A last big warning: The `eval` builtin

```
# This code is evil and should never be used!
function FifthElementOf() {
  local _fifth_array=$1
  eval echo "\"The fifth element is \${$_fifth_array[4]}\""
}
```

```
# Source/define the function above
$ array=(zero one two three four five)
$ FifthElementOf array
The fifth element is four
```

You might be thinking – "It looks OK, isn't it? What's wrong man?"

Advanced, but very informative article which you should spend some time at some point on.

A last big warning: The `eval` builtin

```
# This code is evil and should never be used!
function FifthElementOf() {
  local _fifth_array=$1
  eval echo "\"The fifth element is \${$_fifth_array[4]}\""
}
```

```
# Source/define the function above
$ array=(zero one two three four five)
$ FifthElementOf array
The fifth element is four
```

You might be thinking – "It looks OK, isn't it? What's wrong man?" – Consider then:

```
$ FifthElementOf 'x}"; date; #'
```


Advanced, but very informative article which you should spend some time at some point on.

A last big warning: The `eval` builtin

```
# This code is evil and should never be used!
function FifthElementOf() {
  local _fifth_array=$1
  eval echo "\"The fifth element is \${$_fifth_array[4]}\""
}
```

```
# Source/define the function above
$ array=(zero one two three four five)
$ FifthElementOf array
The fifth element is four
```

You might be thinking – "It looks OK, isn't it? What's wrong man?" – Consider then:

```
$ FifthElementOf 'x}"; date; #'
The fifth element is
Wed 28 Aug 14:43:59 CEST 2019  # AAAAAARRRGGGGHHH!!!!
```

Take-home lesson

An inappropriate use of `eval` can lead to arbitrary code execution!

Advanced, but very informative article which you should spend some time at some point on.

A last big warning: The `eval` builtin

```
# This code is evil and should never be used!
function FifthElementOf() {
  local _fifth_array=$1
  eval echo "\"The fifth element is \${$_fifth_array[4]}\""
}
```

```
# Source/define the function above
$ array=(zero one two three four five)
$ FifthElementOf array
The fifth element is four
```

You might be thinking – "It looks OK, isn't it? What's wrong man?" – Consider then:

```
$ FifthElementOf 'x}"; date; #'
The fifth element is
Wed 28 Aug 14:43:59 CEST 2019  # AAAAAARRRGGGGHHH!!!!
```

What about `rm -rf /` here?

Take-home lesson

An inappropriate use of `eval` can lead to arbitrary code execution!

Advanced, but very informative article which you should spend some time at some point on.

Built in VS external programs

The volcanic beach of the Jökursárlón

Do not underestimate the difference

Internal Commands:

Commands which are built into the shell. This means that the code that implements a builtin is in `/bin/bash`.

External Commands:

When an external command has to be executed, the shell looks for its path given in `PATH` variable and also a new process has to be spawned and the command gets executed.

The precedence order for command names

1. Alias
2. Function { You can shadow builtin commands creating a function with the same name }
3. Builtin { The `builtin` builtin serves to refer to a shadowed builtin → `help builtin` }
4. Keywords
5. External command in the directories listed in `${PATH}` in order.

A trivial benchmark

```
$ mkdir tmpFolder; cd tmpFolder
$ time for((i=0; i<10000; i++)); do echo > $i; done; echo

real    0m18.364s
user    0m0.659s
sys     0m1.600s

$ time for((i=0; i<10000; i++)); do /bin/echo > $i; done

real    0m37.524s
user    0m12.325s
sys     0m8.010s
$ cd ..; rm -r tmpFolder
```

Do not overdo, just keep it in mind

- «Premature optimisation is the root of all evil» – **Donald Knuth**
- **To avoid unnecessary use of external commands is a good practice**

Script autocompletion

A ice-dragon on the Jökursárlón's beach

Programmable completion

A complex world plenty of opportunities: 🔗 Bash manual v5.0 sections 8.6 and 8.7

- Whenever you press `TAB` while typing in your terminal,
  you trigger a series of operations that the shell executes underneath.
- This mechanism is actually programmable and it is usually available system-wide.
- It is nothing but… a bash script to be sourced in your environment!
- It is usually automatically sourced somewhere, at university in the shell configuration file.

```
# In a default ~/.bashrc file at some point:
if [ -f /etc/bash_completion ]; then
    . /etc/bash_completion
fi
```

```
$ cat /etc/bash_completion
. /usr/share/bash-completion/bash_completion
$ wc -l /usr/share/bash-completion/bash_completion
2171
```

- The mechanism is triggered via the `complete` builtin.

The program completion ingredients

`compgen`	Generate possible completion matches for word according to the options and write the matches to the standard output.
`complete`	Specify how arguments should be completed.
`compopt`	Modify completion options for each name according to the options, or for the currently-executing completion if no names are supplied.

`COMP_WORDS`	An array variable consisting of the individual words in the current command line.
`COMP_CWORD`	An index into `${COMP_WORDS}` of the word containing the current cursor position.
`COMPREPLY`	An array variable from which Bash reads the possible completions generated. Each array element contains one possible completion.

There are further relevant variables — `COMP_WORDBREAKS`, `COMP_LINE`, `COMP_POINT`, etc. — refer to the Bash manual v5.0 sections 5.2 .

The standard pattern

1. Make the script for which you want to implement autocompletion executable.
2. Create an autocompletion script **to be sourced** in your environment.
3. Associate a function to the command you want (your script's name) using.

```
complete -F _script_completion  script
```

   After having sourced the autocompletion script, this function will be automatically invoked by the shell when pressing `TAB` after the name of your script.

4. Write the autocompletion function keeping in mind how it works.

A standard invocation

- `$1` is the name of the command whose arguments are being completed;
- `$2` is the word being completed;
- `$3` is the word preceding the word being completed;
- The array variable `COMPREPLY` contains possible completions.

A basic example as starting point

Suppose to have a `measure` script that accepts the following options on the command line:

- `--inputfile` followed by a filename;
- a series of other options, e.g. `--verbose` and `--conservative`.

How do I implement autocompletion for it?

Usually, you want to implement autocompletion for a script that you use very often. It is common to make it executable and remove the `.bash` extension.

A basic example as starting point

Suppose to have a `measure` script that accepts the following options on the command line:

- `--inputfile` followed by a filename;
- a series of other options, e.g. `--verbose` and `--conservative`.

How do I implement autocompletion for it?

```
$ emacs -nw ~/.measure-completion.bash
```


Usually, you want to implement autocompletion for a script that you use very often. It is common to make it executable and remove the `.bash` extension.

A basic example as starting point

```
# You do not need a shebang here! This script will be sourced.
```


Usually, you want to implement autocompletion for a script that you use very often. It is common to make it executable and remove the `.bash` extension.

A basic example as starting point

```
# You do not need a shebang here! This script will be sourced.

if [[ "${BASH_SOURCE[0]}" = "${0}" ]]; then
    printf "\n ERROR: File \"${BASH_SOURCE[0]}\" cannot be executed!\n\n"
    exit 1
fi
```


Usually, you want to implement autocompletion for a script that you use very often. It is common to make it executable and remove the `.bash` extension.

A basic example as starting point

```
# You do not need a shebang here! This script will be sourced.

if [[ "${BASH_SOURCE[0]}" = "${0}" ]]; then
    printf "\n ERROR: File \"${BASH_SOURCE[0]}\" cannot be executed!\n\n"
    exit 1
fi

function _measure_completion()
{
    case "$3" in
        --inputfile )
            COMPREPLY=( $(compgen -f -- "$2") )
            ;;
        * )
            COMPREPLY=(
                $(compgen -W "--inputfile --verbose --conservative" -- "$2")
            )
            ;;
    esac
}

complete -F _measure_completion   measure
```


Usually, you want to implement autocompletion for a script that you use very often. It is common to make it executable and remove the `.bash` extension.

A basic example as starting point

```
# You do not need a shebang here! This script will be sourced.

if [[ "${BASH_SOURCE[0]}" = "${0}" ]]; then
    printf "\n ERROR: File \"${BASH_SOURCE[0]}\" cannot be executed!\n\n"
    exit 1
fi

function _measure_completion()
{
    case "$3" in
        --inputfile )
            COMPREPLY=( $(compgen -f -- "$2") )
            ;;
        * )
            COMPREPLY=(
                $(compgen -W "--inputfile --verbose --conservative" -- "$2")
            )
            ;;
    esac
}

complete -F _measure_completion   measure
```

What is bad in this implementation?

Usually, you want to implement autocompletion for a script that you use very often. It is common to make it executable and remove the `.bash` extension.

A basic example as starting point

```
# You do not need a shebang here! This script will be sourced.

if [[ "${BASH_SOURCE[0]}" = "${0}" ]]; then
    printf "\n ERROR: File \"${BASH_SOURCE[0]}\" cannot be executed!\n\n"
    exit 1
fi

function _measure_completion()
{
    case "$3" in
        --inputfile )
            COMPREPLY=( $(compgen -f -- "$2") )
            ;;
        * )
            COMPREPLY=(
                $(compgen -W "--inputfile --verbose --conservative" -- "$2")
            )
            ;;
    esac
}

#complete -F _measure_completion   measure
complete -o filenames -F _measure_completion   measure
```

What is still bad in this implementation?

Usually, you want to implement autocompletion for a script that you use very often. It is common to make it executable and remove the `.bash` extension.

A basic example as starting point

Suppose to have a `measure` script that accepts the following options on the command line:

- `--inputfile` followed by a filename;
- a series of other options, e.g. `--verbose` and `--conservative`.

How do I implement autocompletion for it?

```
$ emacs -nw ~/.measure-completion.bash
$ . ~/.measure-completion.bash
$ ls
file01.dat   file02.dat   file03.dat   backup
$ ./measure <TAB>
$ ./measure --<TAB-TAB>
--conservative  --inputfile     --verbose
$ ./measure --inputfile <TAB-TAB>
backup/   file01.dat   file02.dat   file03.dat
# There is room for improvement!
$ ./measure --inputfile file02.dat <TAB-TAB>
--conservative  --inputfile     --verbose
```


Usually, you want to implement autocompletion for a script that you use very often. It is common to make it executable and remove the `.bash` extension.

GNU Readline

A trail at sunset in the Reykjanesfólkvangur Reserve

The GNU Readline Library

Official website

What is it for?

The GNU Readline library provides a set of functions for use by applications that allow users to edit command lines as they are typed in. Both Emacs and vi editing modes are available. The Readline library includes additional functions to maintain a list of previously-entered command lines, to recall and perhaps reedit those lines.

The GNU Readline Library

Official website

What is it for?

The GNU Readline library provides a set of functions for use by applications that allow users to edit command lines as they are typed in. Both Emacs and vi editing modes are available. The Readline library includes additional functions to maintain a list of previously-entered command lines, to recall and perhaps reedit those lines.

- Documentation can be also found in the Bash manual v5.0 sections 8.1 to 8.5 .
- It provide lots of functionality, some of which you might already be used to.
- If you use bash in your terminal, take advantage of it, please!
- A great deal of run-time behaviour is changeable with 45 variables. { Section 8.3.1 }
- There are ~170 Readline commands that may be bound to key sequences. { Section 8.4 }

The Readline init file

- The Readline library comes with a set of Emacs-like keybindings installed by default.
- It is possible to use a different set of keybindings, though.
- Any user can customise programs that use Readline by putting commands in an inputrc file, conventionally in his home directory.
- The name of this file is taken from the value of the shell variable `INPUTRC`. If that variable is unset, the default is `~/.inputrc`. If that file does not exist or cannot be read, the ultimate default is `/etc/inputrc`.
- When a program which uses the Readline library starts up, the init file is read, and the key bindings are set.
- The `CTRL-x CTRL-r` command re-reads this init file, thus incorporating any changes that you might have made to it.

The `bind` builtin

Interacting with the Readline library

```
$ help bind

bind: bind [-lpsvPSVX] [-m keymap] [-f filename] [-q name] [-u name]
           [-r keyseq] [-x keyseq:shell-command]
           [keyseq:readline-function or readline-command]

    Set Readline key bindings and variables.

    Bind a key sequence to a Readline function or a macro, or set a
    Readline variable. The non-option argument syntax is equivalent to
    that found in ~/.inputrc, but must be passed as a single argument:
    e.g., bind '"\C-x\C-r": re-read-init-file'.
```

The `bind` builtin

Interacting with the Readline library

```
$ help bind

[...]

 Options:

   -l                 List names of functions.
   -P                 List function names and bindings.
   -p                 List functions and bindings in a form that can be
                      reused as input.
   -S                 List key sequences that invoke macros and their values
   -s                 List key sequences that invoke macros and their values
                      in a form that can be reused as input.
   -V                 List variable names and values
   -v                 List variable names and values in a form that can
                      be reused as input.
   -q  function-name  Query about which keys invoke the named function.
   -u  function-name  Unbind all keys which are bound to the named function.
   -r  keyseq         Remove the binding for KEYSEQ.
   -f  filename       Read key bindings from FILENAME.
   # Plus few more
```

An extremely useful example

An extremely useful example

Warning: Once you try it, you'll never go back!

An extremely useful example

If you want to affect bash only

You can put the following block in your `~/.bashrc` file and use then the `bind` builtin.

```
#If session is interactive do some useful binding
if [[ -t 1 ]]; then
    #To bind page-up and page-down to search in history
    # (command starting by what typed)
    bind '"\e[5~": history-search-backward'
    bind '"\e[6~": history-search-forward'
    #To bind up and down arrow to search in history
    # (anything containing what typed)
    bind '"\e[A": history-substring-search-backward'
    bind '"\e[B": history-substring-search-forward'
    #Move right and left one word with CTRL+→ and CTRL+←
    bind '"\e[1;5D": backward-word'
    bind '"\e[1;5C": forward-word'
fi
```

An extremely useful example

You can act globally instead

To affect all programs that use the GNU Readline library, you can put the following block in your `~/.inputrc` file and load it.

```
#To use global settings as well
$include /etc/inputrc

#To bind page-up and page-down to search in history
# (command starting by what typed)
"\e[5~": history-search-backward
"\e[6~": history-search-forward
#To bind up and down arrow to search in history
# (anything containing what typed)
"\e[A": history-substring-search-backward
"\e[B": history-substring-search-forward
#Move right and left one word with CTRL+→ and CTRL+←
"\e[1;5D": backward-word
"\e[1;5C": forward-word
```


Remember, you can reload the Readline init file via `CTRL-x CTRL-r` or via `bind -f ~/.inputrc` (removed commands will not be cleaned).

An extremely useful example

No more need of `CTRL-r`

You can try to type something and press the arrow up and see how it is.

```
#To use global settings as well
$include /etc/inputrc

#To bind page-up and page-down to search in history
# (command starting by what typed)
"\e[5~": history-search-backward
"\e[6~": history-search-forward
#To bind up and down arrow to search in history
# (anything containing what typed)
"\e[A": history-substring-search-backward
"\e[B": history-substring-search-forward
#Move right and left one word with CTRL+→ and CTRL+←
"\e[1;5D": backward-word
"\e[1;5C": forward-word
```


Remember, you can reload the Readline init file via `CTRL-x CTRL-r` or via `bind -f ~/.inputrc` (removed commands will not be cleaned).