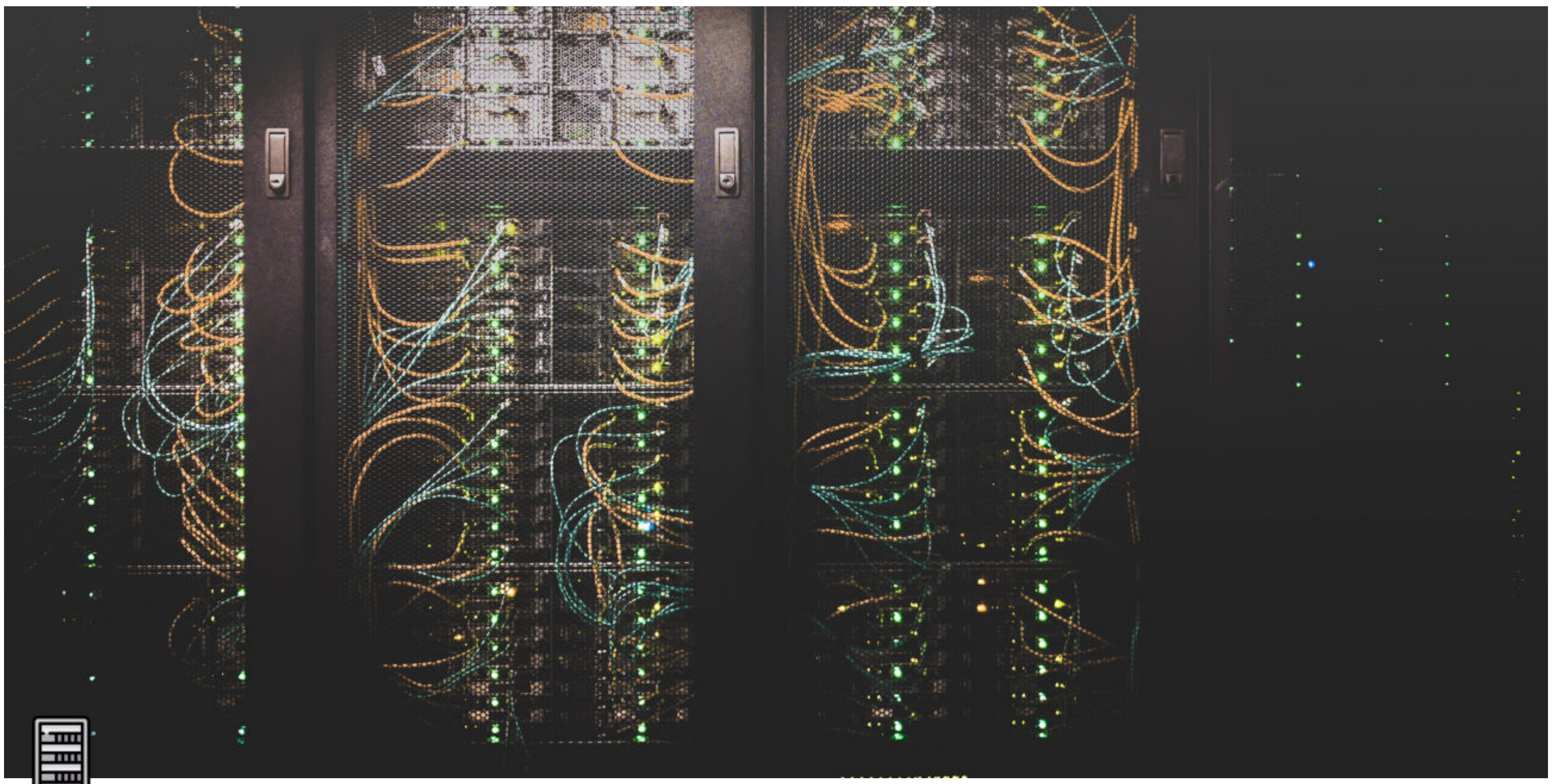


Combining commands and files

Type	📒 Lecture
Date	@January 9, 2022
Lecture #	1
Lecture URL	https://youtu.be/Lcx9UsS7y8Y
Notion URL	https://21f1003586.notion.site/Combining-commands-and-files-2476c1091a704743840ae8b76ab078c9
Week #	3

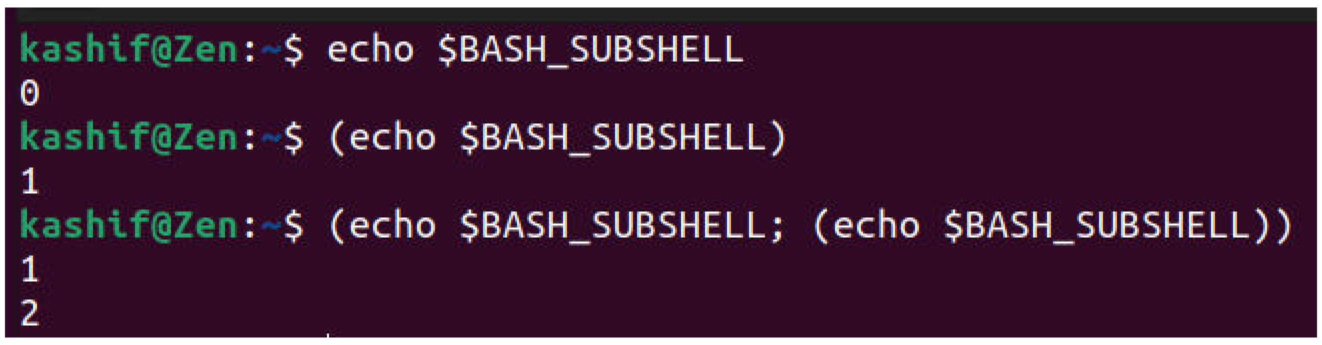
Executing multiple commands

- `command1; command2; command3`
    - Each command will be executed one after the other
- `command1 && command2 && command3`
    - This works as a logical AND
    - The subsequent commands after `command-n` will not run if the previous command resulted in an error
- `command1 || command2 || command3`
    - This works as a logical OR
    - The subsequent commands after `command-n` will not run if the previous command resulted in a success

`(<command>)`

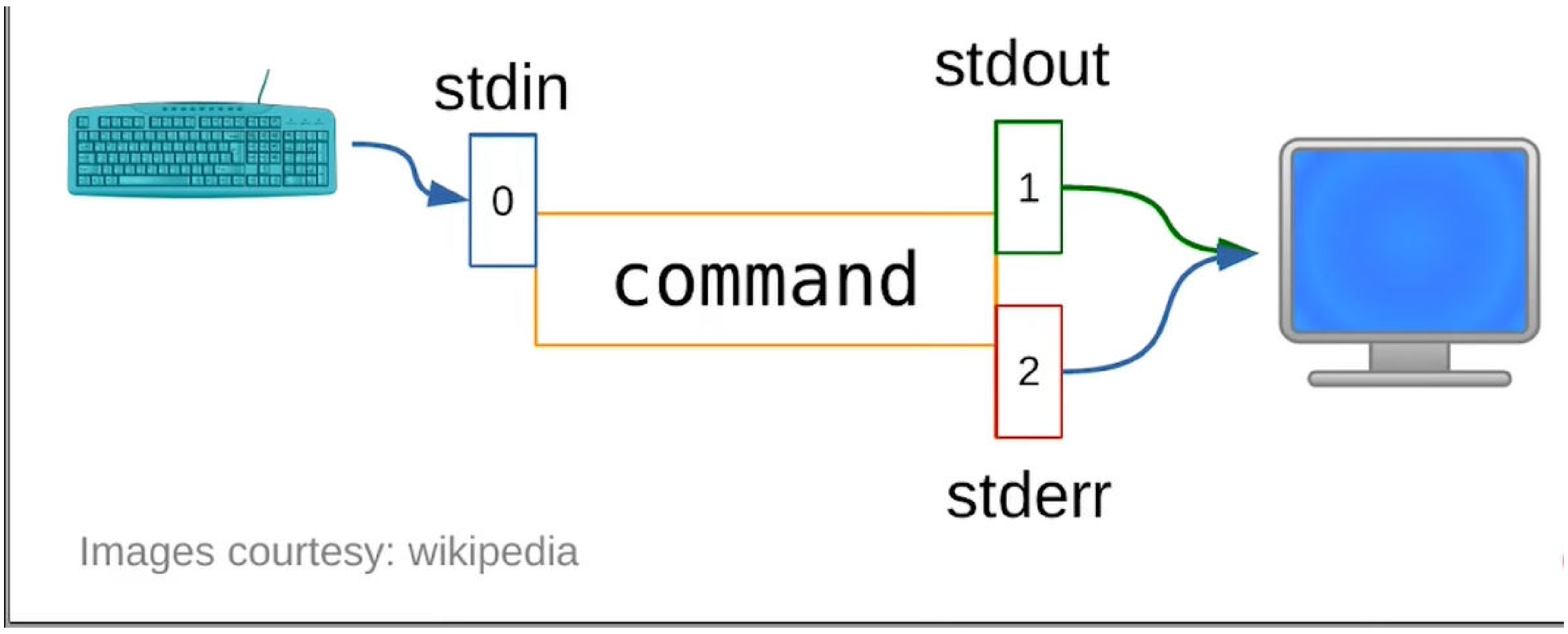
We can run any command enclosed within parentheses to execute them in a subshell, and returned back the result

We can execute a subshell within a subshell too



```
kashif@Zen:~$ echo $BASH_SUBSHELL
0
kashif@Zen:~$ (echo $BASH_SUBSHELL)
1
kashif@Zen:~$ (echo $BASH_SUBSHELL; (echo $BASH_SUBSHELL))
1
2
```

File descriptors

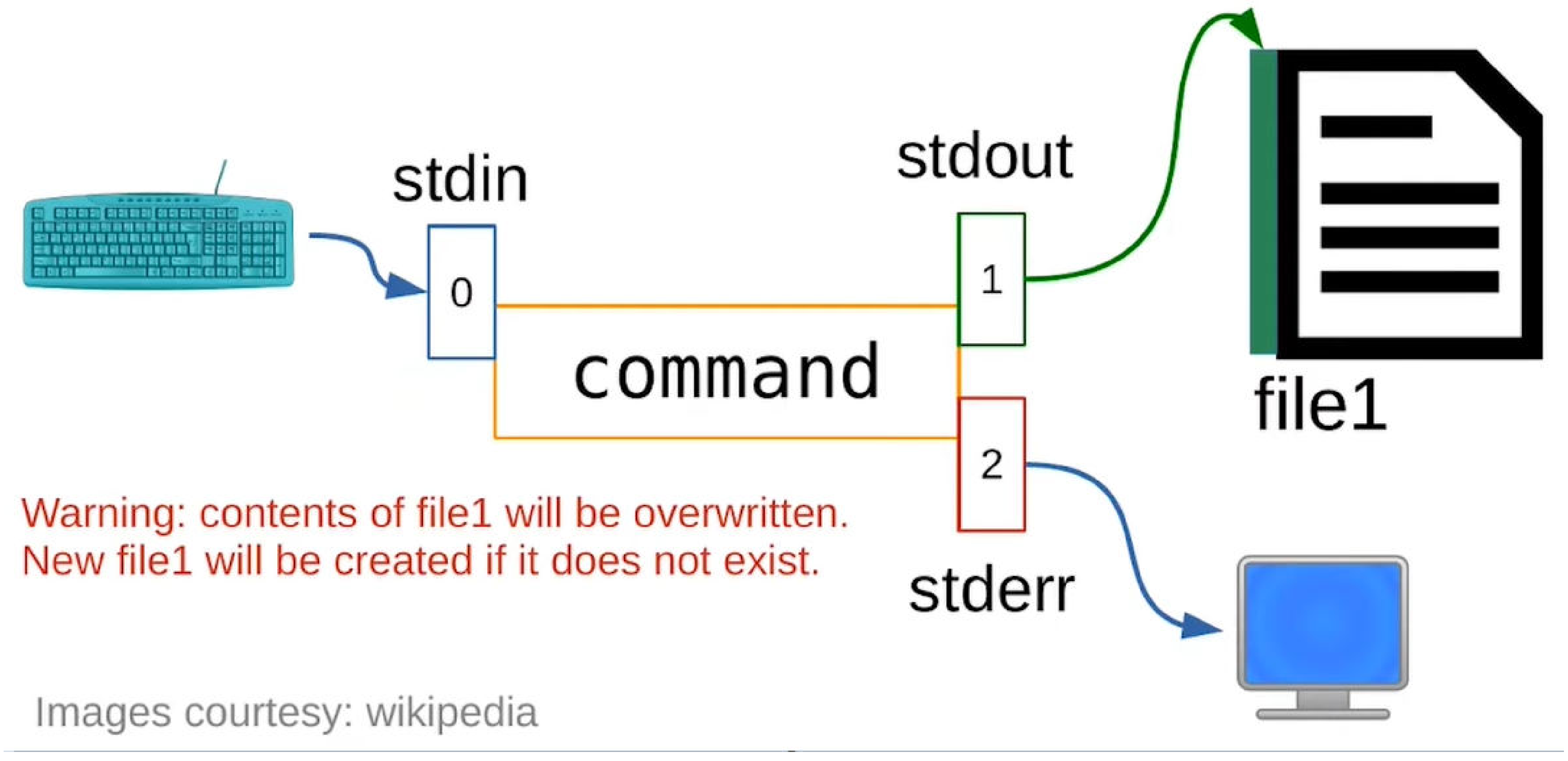


Every command in Linux has 3 file descriptors

- `stdin` (0)
  - It is a pointer to a stream that is coming from the keyboard (or the user input)
- `stdout` (1)
  - Points to the screen where the output is made
- `stderr` (2)
  - Points to the screen where the output is made

`command > file1`

- The output of the `command` should be written to `file1`


stdin
0
command
stdout
1
2
stderr
file1
Warning: contents of file1 will be overwritten.
New file1 will be created if it does not exist.
Images courtesy: wikipedia

Create a file using `cat` command

`cat > filename`

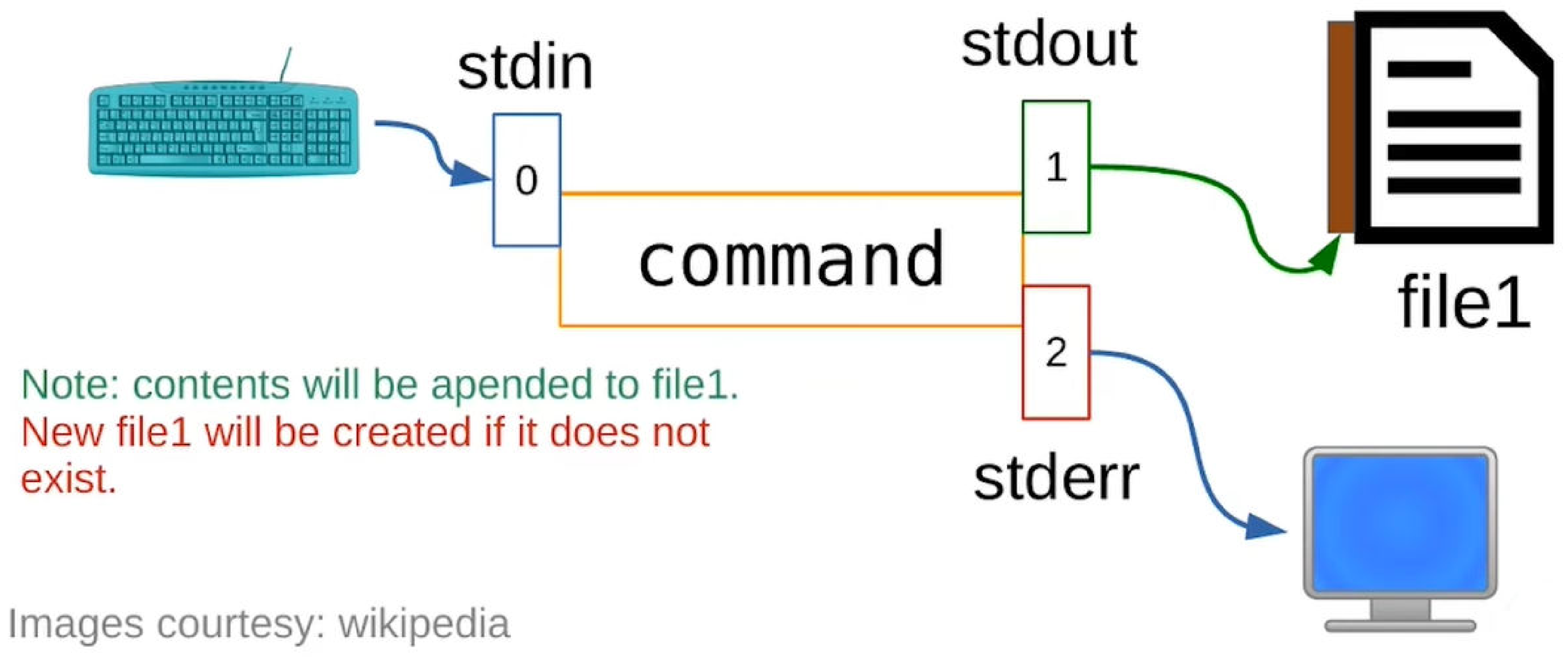
When we type this command, the `cat` command is supposed to receive the input from a file that is listed in the command line, but instead, we left that intentionally blank

So, the `cat` command, instead, reads the content from the `stdin`, i.e. the keyboard

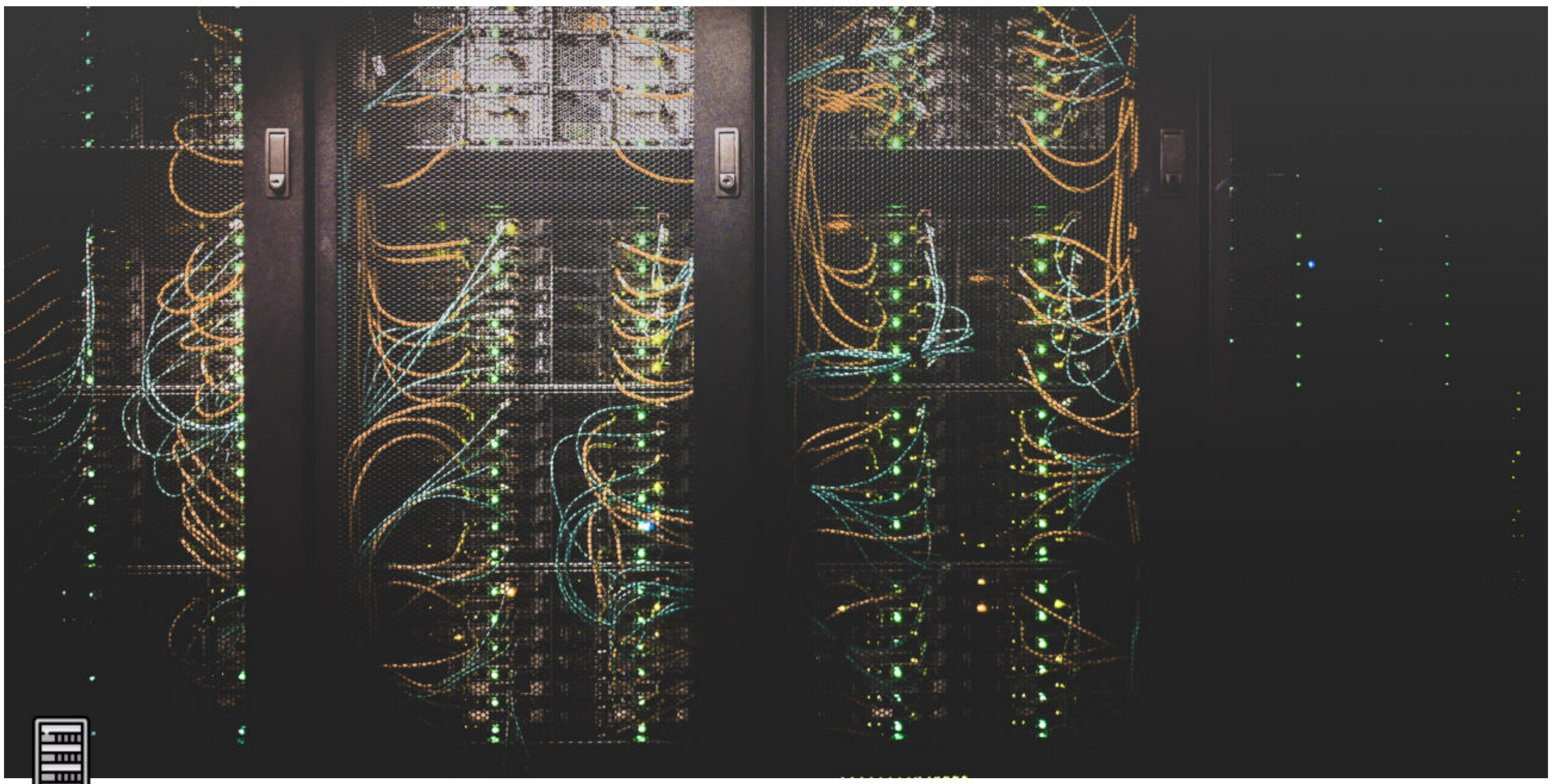
To exit, press `Ctrl + D`

`command >> file1`

- The output of `command` will be appended to `file1`


stdin
0
stdout
1
command
2
stderr
file1
Note: contents will be apended to file1.
New file1 will be created if it does not exist.
Images courtesy: wikipedia

Similarly, we can use `>>` instead of `>` while creating a new file using the `cat` command

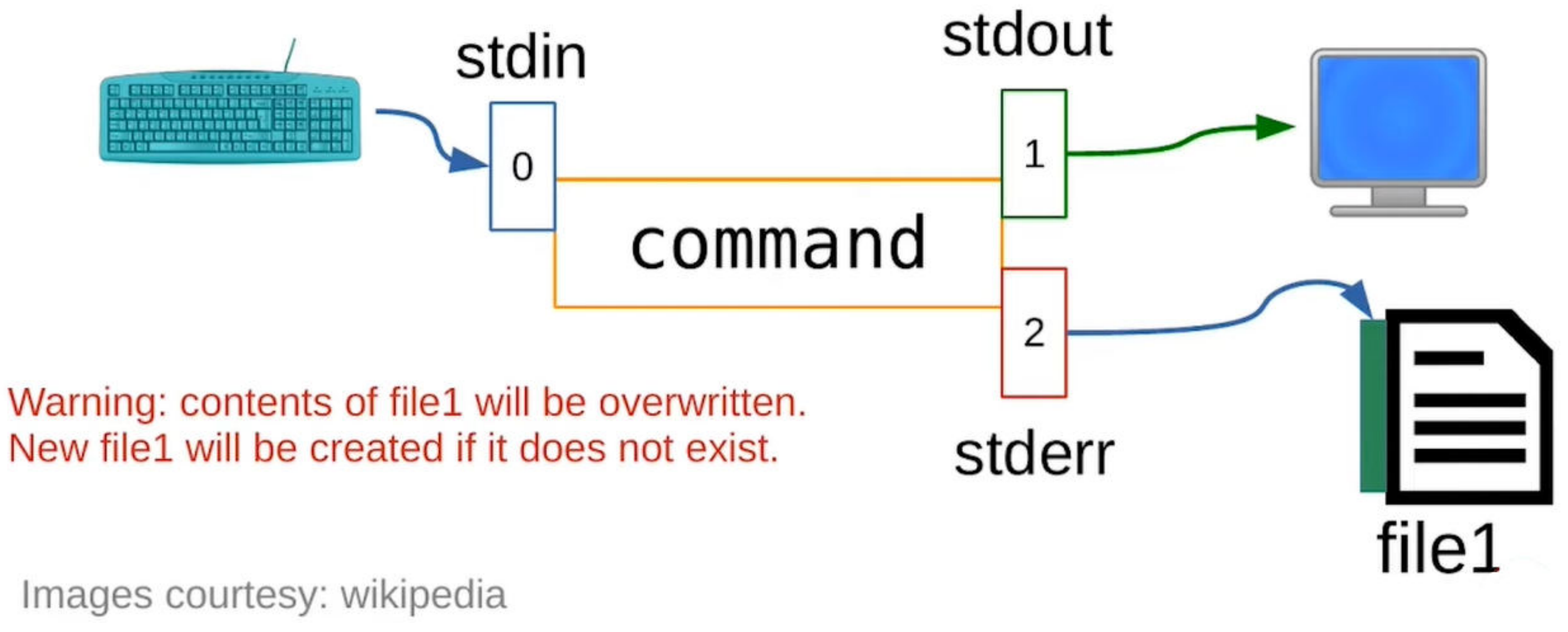


Redirections

Type	📒 Lecture
Date	@January 9, 2022
Lecture #	2
Lecture URL	https://youtu.be/BBh69kH_G_Y
Notion URL	https://21f1003586.notion.site/Redirections-734673f36f21448f99de25ccb092c8d4
Week #	3

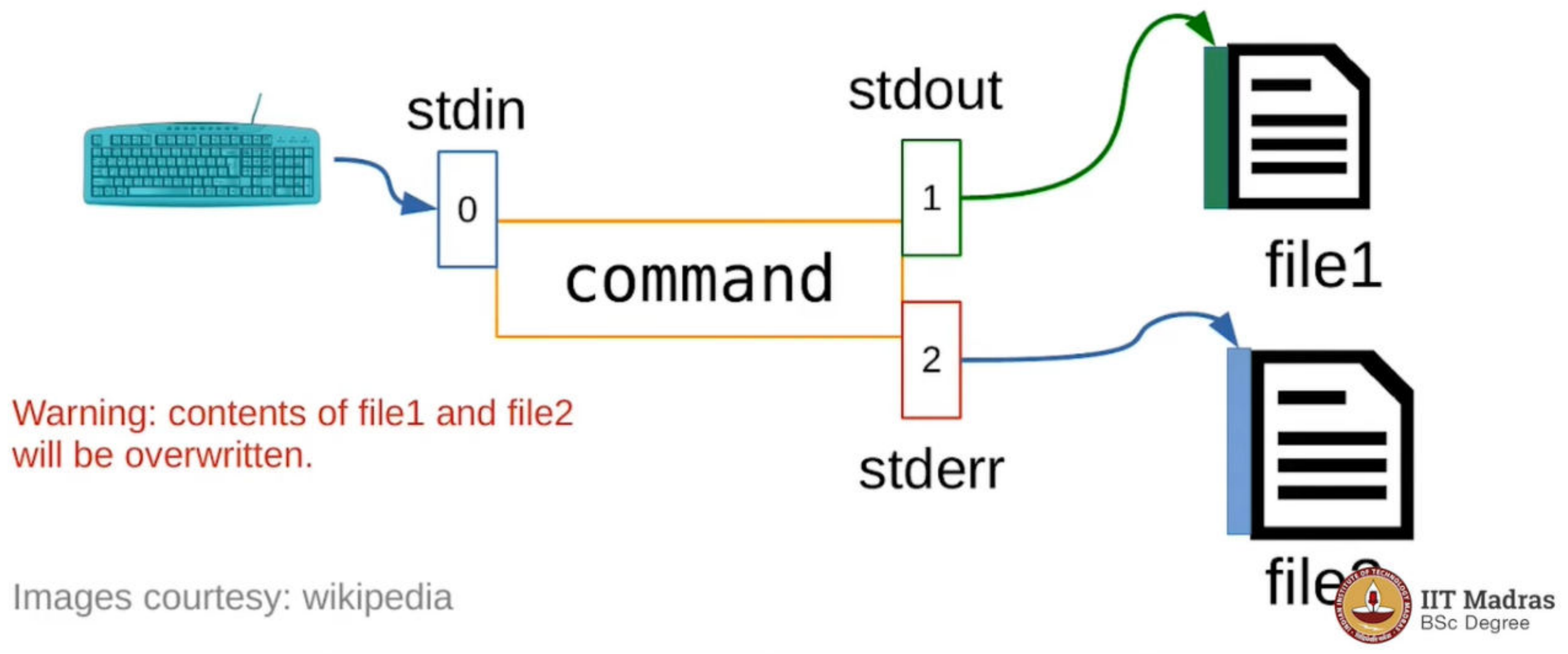
`command 2> file1`

- Redirect the output of the `command` to `stdout`, which is the display in this case
- Redirect the error of the `command` to `file1`


stdin
stdout
0
1
command
2
stderr
Warning: contents of file1 will be overwritten.
New file1 will be created if it does not exist.
file1
Images courtesy: wikipedia

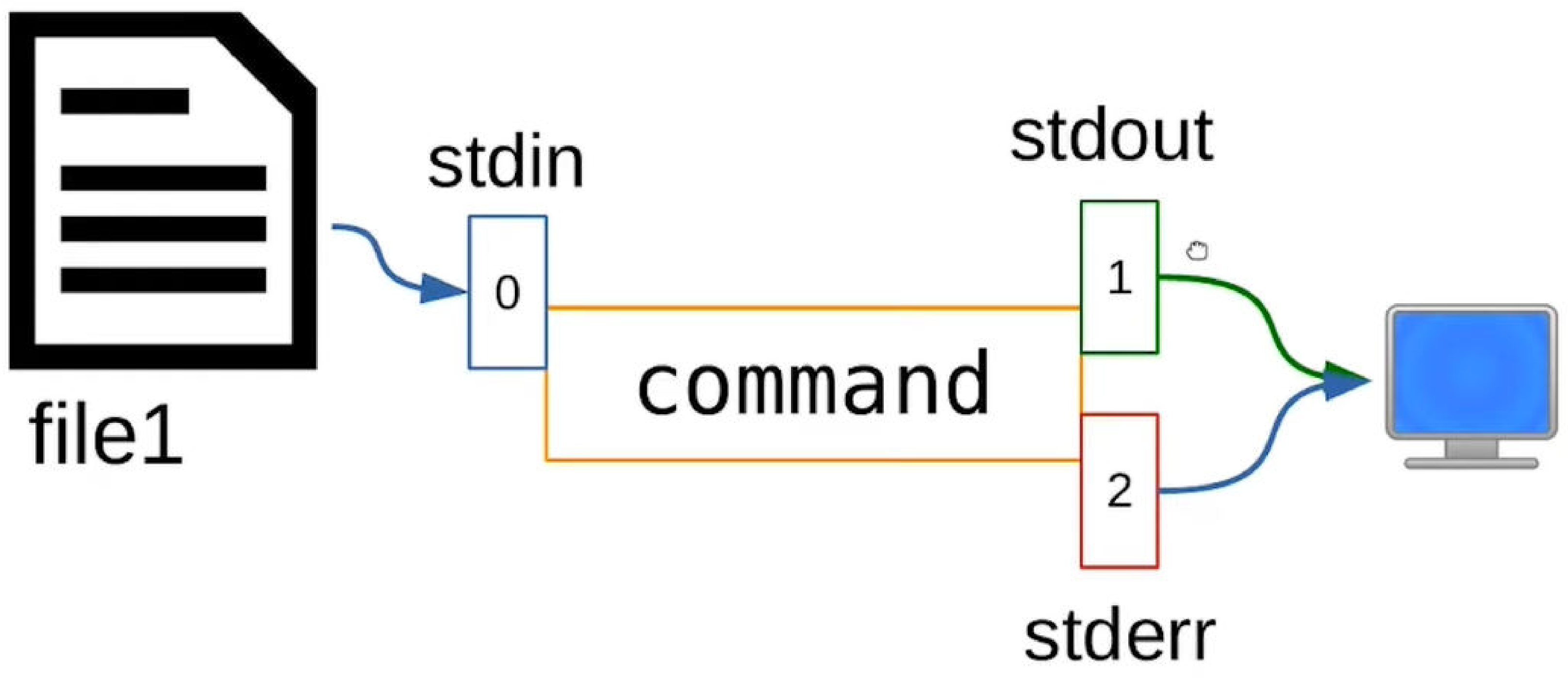
`command > file1 2> file2`

- Redirect the output of the `command` to the `stdout`, i.e. `file1`
- Redirect the error of the `command` to `stderr`, i.e. `file2`


stdin
0
stdout
1
command
2
stderr
file1
file2
Warning: contents of file1 and file2 will be overwritten.
Images courtesy: wikipedia
IIT Madras
BSc Degree

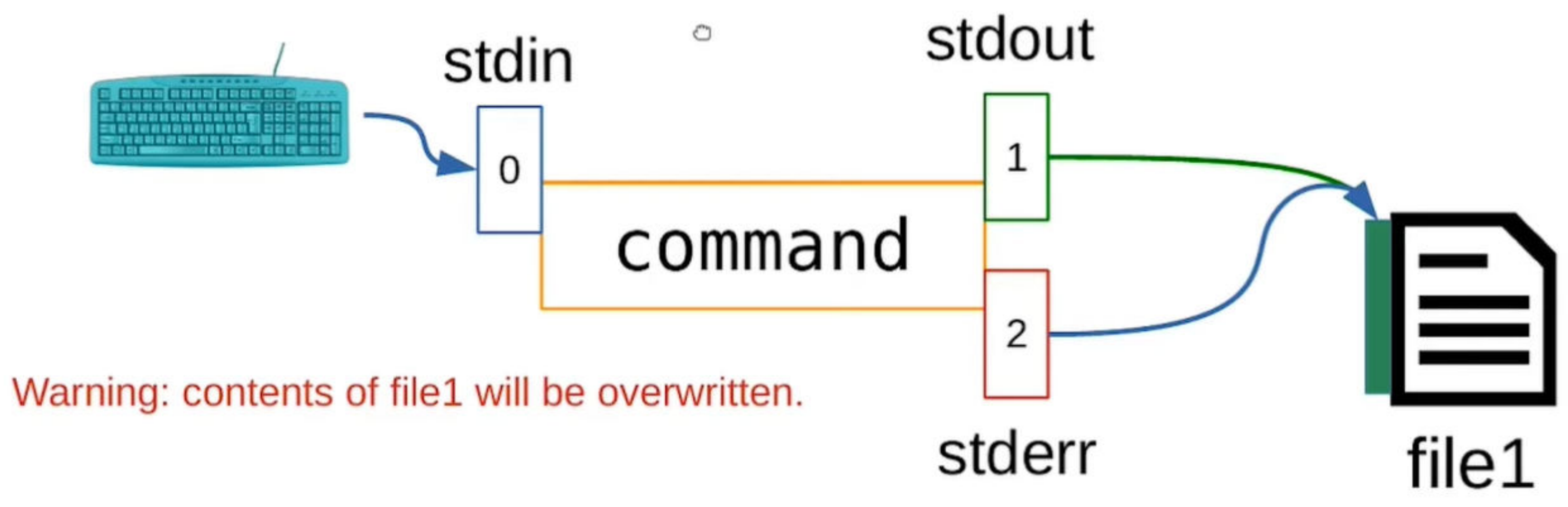
`command < file1`

- Any `command` which takes input from the keyboard, now takes input from `file1`



`command > file1 2>&1`

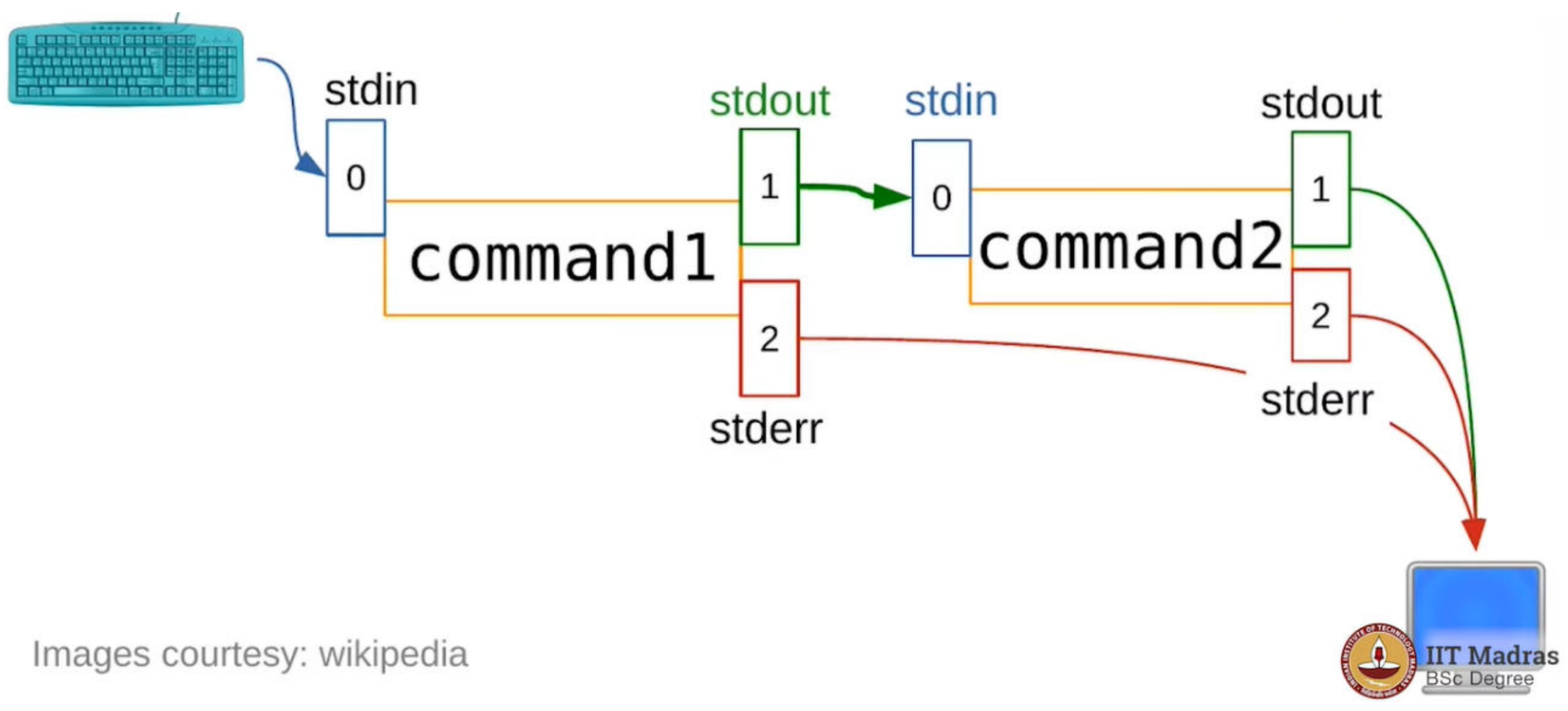
- The output of `command` is written to `file1`
- The error is redirected to stream 1, which is `stdout`



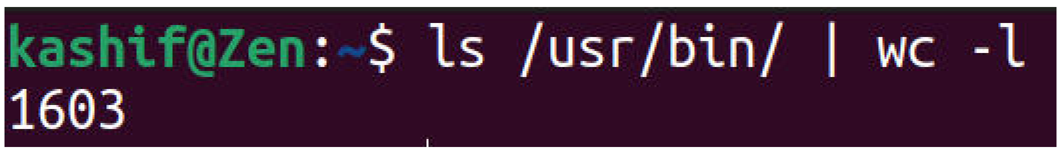
`command1 | command2` → **pipe operator** `|`

- The output of `command1` is sent to `command2` as input

- By default, the `stderr` will output to the display



Count the number of files in the directory `/usr/bin`



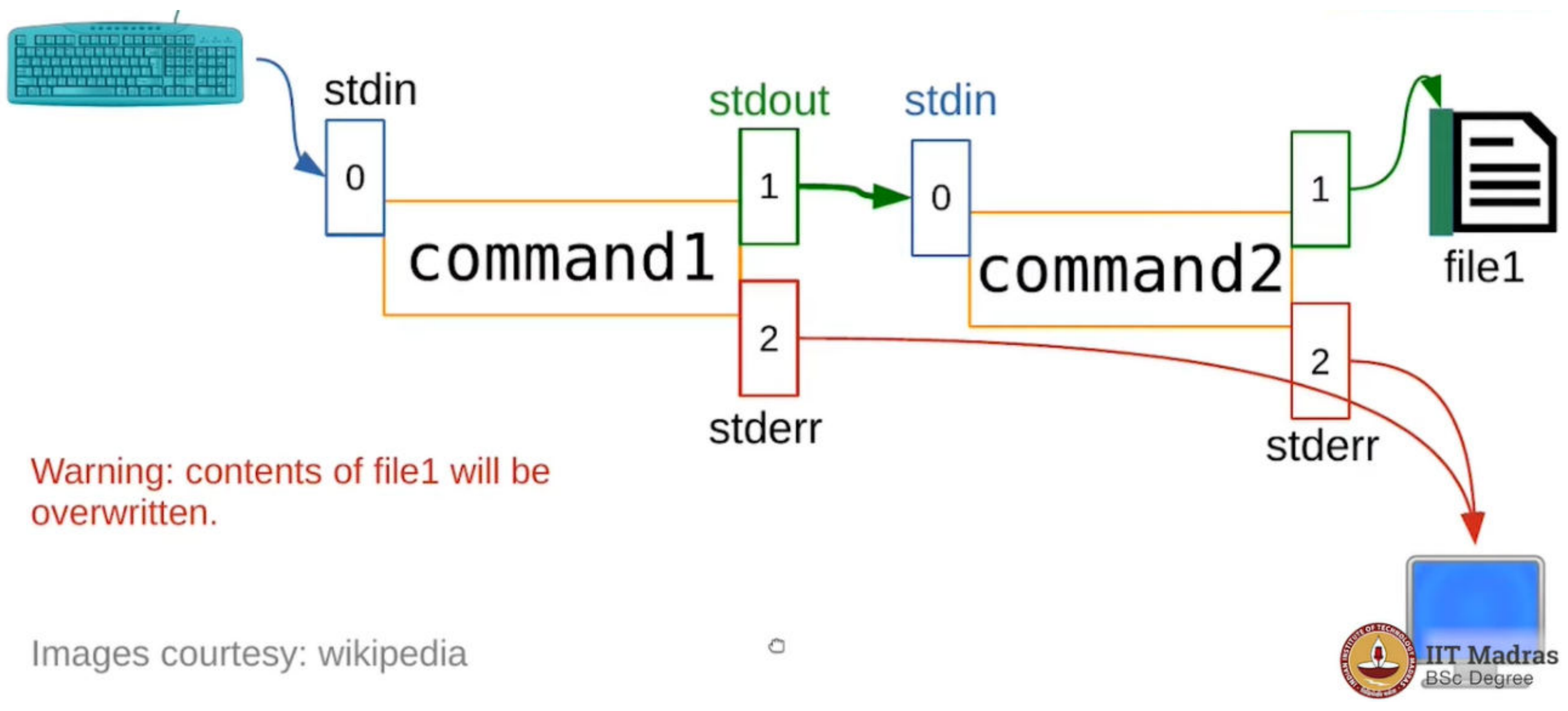
```
kashif@Zen:~$ ls /usr/bin/ | wc -l
1603
```

List the files of `/usr/bin` directory, but use the `less` command to scroll at ease

`ls /usr/bin | less`

`command1 | command2 > file1`

- The `stdout` of `command1` is mapped to `stdin` of `command2`
- The `stdout` of `command2` is written to `file1`
- The `stderr` is output to the display



`/dev/null`

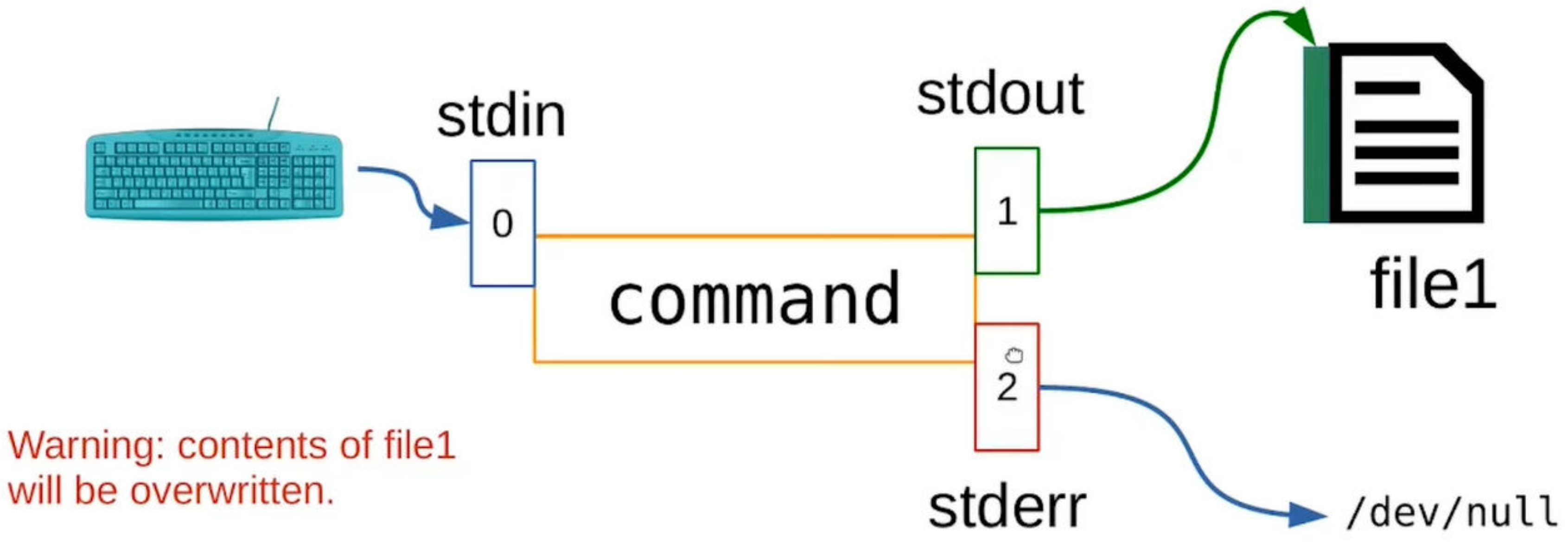
- A sink for output to be discarded
- Use → silent and clean scripts

So, a typical usage looks like ...

`command > file1 2> /dev/null`

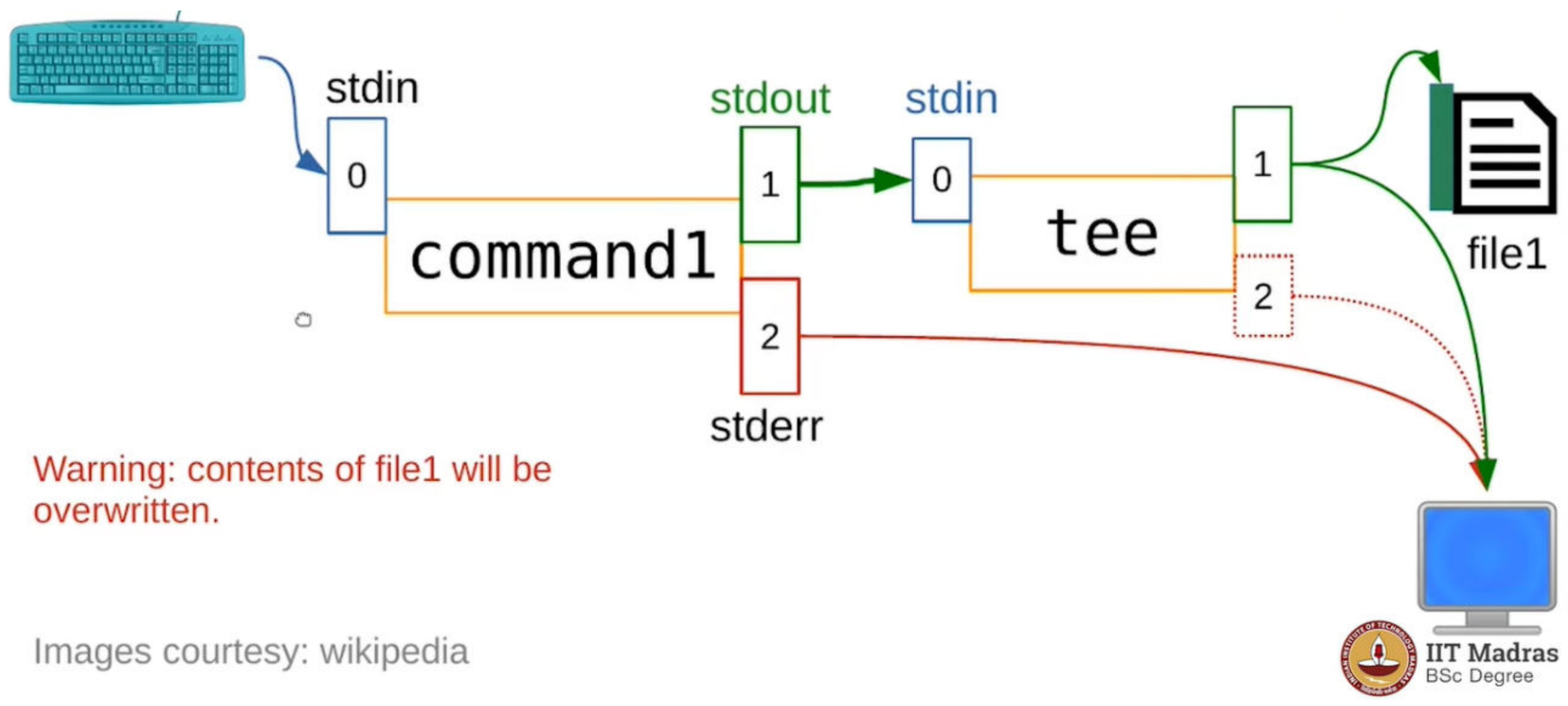
- The output of `command` is written to `file1`

- The `stderr` is written to `/dev/null`, ~~*which gets warped to another dimension*~~



`command1 | tee file1`

- The `tee` command splits the output into 2 streams, one stream is written to the `file1` another, one to the display
    - This command can write to multiple files as well

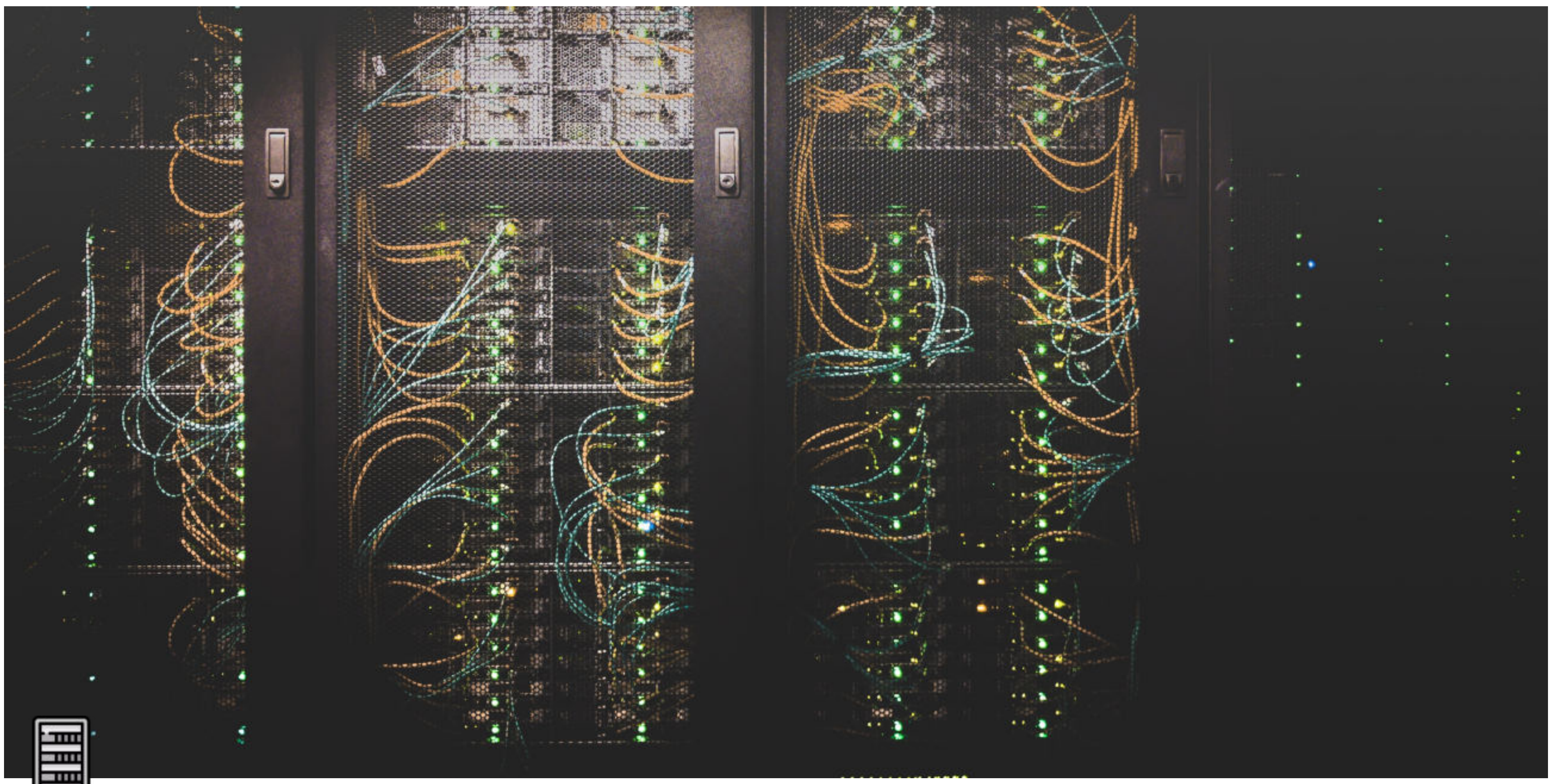

stdin
0
command1
stdout
1
2
stderr
stdin
0
tee
1
2
file1
Warning: contents of file1 will be overwritten.
Images courtesy: wikipedia
IIT Madras
BSc Degree

`diff` command

- This command compares files line-by-line

Usage

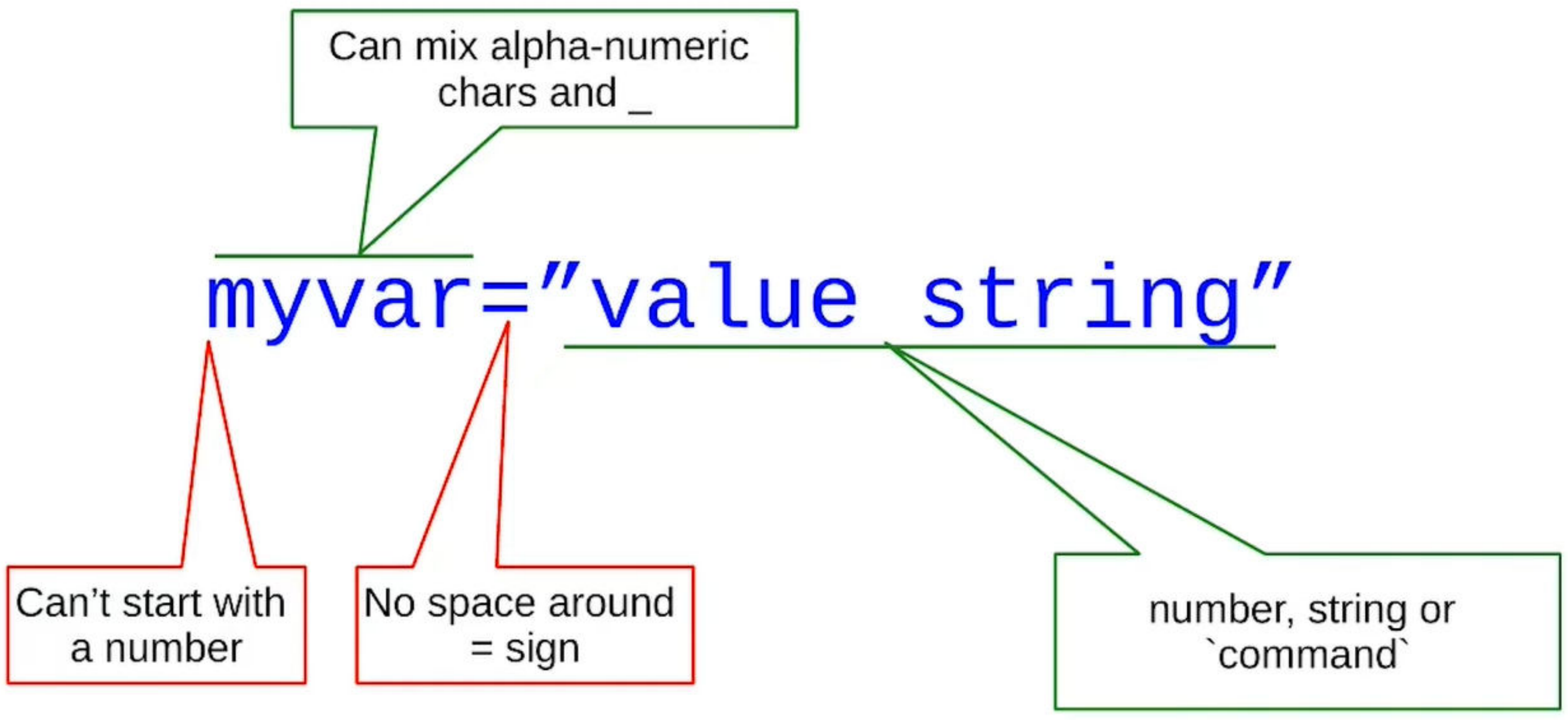
`diff file1 file2`



Shell variables

Type	📒 Lecture
Date	@January 9, 2022
Lecture #	3
Lecture URL	https://youtu.be/QX5XElFRpck
Notion URL	https://21f1003586.notion.site/Shell-variables-7fc4b392e99b4d59a13eebefdae88e1e
Week #	3

Creating a variable


Can mix alpha-numeric chars and _
myvar="value string"
Can't start with a number
No space around = sign
number, string or `command`

Exporting a variable

Exporting means making the value of the variable available to a shell spawned by the current shell (wut)

`export myvar="value string"`

OR

`myvar="value string"`

`export myvar`

Using variable values

`echo $myvar`

`echo ${myvar}`

`echo “${myvar}”`

Removing a variable

`unset myvar`

Removing value of a variable

`myvar=`

Test if a variable is set

`[[ -v myvar ]];`

`echo $?`

Return codes:

0 → success (variable `myvar` is set)

1 → failure (variable `myvar` is not set)

Test if a variable is *not* set

`[[ -z ${myvar+x} ]];`

Here, x can be any string

`echo $?`

Return codes:

0 → success (variable myvar is not set)

1 → failure (variable myvar is set)

Substitute default value

If the variable `myvar` is not set, use `“default”` as its default value

`echo ${myvar:-”default”}`

So, if `myvar` is set, display its value else display “default”

Reset value if variable is set

If the variable `myvar` is set, then set `“default”` as its value

`echo ${myvar:+”default”}`

So, if `myvar` is set, change it’s value to “default” and display it else display null

List of variable names

`echo ${!H*}`

List of names of shell variables that start with `H`

Length of string value

`echo ${#myvar}`

Display length of the string value of the variable `myvar`

If `myvar` is not set, display 0

Slice of a string value

`echo ${myvar:5:4}`

Display 4 chars of the string value of the variable `myvar`, skipping first 5 chars

Remove matching pattern

`echo ${myvar#pattern}` → *match once*

`echo ${myvar##pattern}` → *match max possible*

Keep matching pattern

`echo ${myvar%pattern}` → *match once*

`echo ${myvar%%pattern}` → *match max possible*

Replace matching pattern

`echo ${myvar/pattern/string}` → *match once and replace with* `string`

`echo ${myvar//pattern/string}` → *match max possible and replace with* `string`

Replace matching pattern by location

`echo ${myvar/#pattern/string}` → *match at beginning and replace with* `string`

`echo ${myvar/%pattern/string}` → *match at the end and replace with* `string`

Changing case

`echo ${myvar,}` → *change the first char to lower case*

`echo ${myvar,,}` → *change all chars to lower case*

`echo ${myvar^}` → *change first char to upper case*

`echo ${myvar^^}` → *change all chars to upper case*

Restricting value types

`declare -i myvar` → *only integers can be assigned*

`declare -l myvar` → *only lower case chars can be assigned*

`declare -u myvar` → *only upper case chars can be assigned*

`declare -r myvar` → *variable is read-only*

You can remove the restrictions by replacing the `-` *sign with a* `+` *sign*

However, `declare +r myvar` will ***NOT*** work

Indexed arrays

`declare -a arr` → *declare* `arr` *as an indexed array*

`$arr[0]="value"` → *set value of element with index 0 in the array*

`echo ${arr[0]}` → *value of the element at index 0 of array*

`echo ${#arr[@]}` → *number of elements in the array*

`echo ${!arr[@]}` → *display all the indices used*

`echo ${arr[@]}` → *display values of all elements of the array*

`unset 'arr[2]'` → *delete element with index 2 in the array*

`arr+=("value")` → *append an element with a value to the end of the array*

Associative arrays

Kind of like Hash maps?

`declare -A hash` → *declare* `hash` *as an associative array*

`$hash["a"]="value"` → *set value of element with index "a" in the array*

`echo ${hash["a"]}` → *value of element with index (or key?) "a" in the array*

`echo ${#hash[@]}` → *number of elements in the array*

`echo ${!hash[@]}` → *display all indices used*

`echo ${hash[@]}` → *display values of all elements of the array*

`unset 'hash["a"]'` → *delete element with index "a" in the array*