Redirection: Examples

- Stdout redirected to file

```
find . -name pippo > find-output.txt
```

- Stderr redirected to file

```
find . -name pippo 2> find-errors.txt
```

- discards any errors that are generated by the find command

```
find / -name "*" -print 2> /dev/null
```

/dev/null is a simple device (implemented in software and not corresponding to any hardware device on the system).

/dev/null looks empty when you read from it.

Writing to /dev/null does nothing: data written to this device simply "disappear."

Often a command's standard output is silenced by redirecting it to /dev/null, and this is perhaps the null device's commonest use in shell scripting:

```
command > /dev/null
```

- Redirect both stdout and stderr to file

```
find .  -name pippo &> out_and_err.txt
```

- Redirect stderr to stdout: `find .  -name filename 2>&1`
- Redirect stdout to stderr: `find .  -name filename 1>&2`

Special characters (3)

Pipe [|]. Passes the output (stdout) of a previous command to the input (stdin) of the next one, or to the shell. This is a method of chaining commands together.

```
echo ls -l | bash
#  Passes the output of "echo ls -l" to the shell,
# with the same result as a simple "ls -l".
```

```
cat *.lst | sort | uniq
# Merges and sorts all ".lst" files, then deletes duplicate lines.
```

A pipe sends the stdout of one process to the stdin of another. In a typical case, a command, such as cat or echo, pipes a stream of data to a command that transforms it in input for processing:

```
cat $filename1 $filename2 | grep $search_word
```

Redirection with pipe and tee examples

Examples of redirection of the output of a command to be used as input of another:

- Display the output of a command (in this case ls) by pages:
  ls -la | less
- Count files in a directory:
  ls -l | wc -l
- Count the number of rows containing of the word “canadesi” in the file vialactea.txt
  grep canadesi vialactea.txt | wc -l
- Count the number of words in the rows containing the word “canadesi”

`tee` is useful to redirect output both to stdout and to a file. Example:
find . -name filename.ext 2>&1 | tee -a log.txt
This will take stdout and append it to log file. The stderr will then get converted to stdout which is piped to tee which appends it to the log and sends it to stdout which will either appear on the tty or can be piped to another command.

To go deep: https://stackoverflow.com/questions/2871233/write-stdout-stderr-to-a-logfile-also-write-stderr-to-screen

Exercise: redirection

Create a directory and file tree like this one:

my_examples /ex1.dir
/ex2.txt
/ex3.dir
/*ex3.dir/*file1.txt
/*ex3.dir/*file2.txt
/*ex3.dir/*file3.txt

Remove read permissions to directory */ex1.dir*
Redirect output on a file. Error is displayed on terminal
Redirect error on a file. Output is displayed on terminal
Verify the content of the files
Stderr redirected to file
Redirect output and errors simultaneously

Use pipe to redirect the output of a command to another command and to a file
Use tee to redirect output both to stdout and to a file

UNIX Variables

★ Variables are how programming and scripting languages represent data. A variable is a label, a name assigned to a location holding data.

★ Standard UNIX variables are split into two categories:

- environment variables:
  if set at login, are valid for the duration of the session
- shell variables:
  apply only to the current instance of the shell and are used to set short-term working conditions;

By convention, environment variables have UPPER CASE and shell variables have lower case names.

★ Environment variables are a way of passing information from the shell to programs when you run them. Programs look "in the environment" for variables and if found, will use the values stored.

★ Variables can be set: by the system, by you, by the shell, by any program that loads another program.

bash variables

Variable in bash are untyped.

★ Bash variables are character strings: can contain a number, a character, a string of characters.

★ Depending on context, bash permits arithmetic operations and comparisons on variables. The determining factor is whether the value of a variable contains only digits or not.

★There is no need to declare a variable, just assigning a value to its reference will create it.

bash variables: assignment (1)

It must distinguish between the name (left value) of a variable and its value (right value).

If variable1 is the name of a variable,
then $variable1 is a reference to its value, i.e. the data item it contains.

$variable1 is actually a simplified form of ${variable1}. In contexts where the $variable syntax causes an error, the longer form ${variable} may work.

Referencing (retrieving) the variable value is called variable substitution.

=> No space permitted on either side of = sign when initializing variables.

Example:

```
a=375      # Initialize variable
hello=$a   # No space permitted on either side of = sign when initializing variables.
#     ^ ^
# What happens if there is a space? Bash will treat the variable name as a program to
# execute, and the = as its first parameter. TRY
#
echo hello       # hello ## Not a variable reference, just the string "hello" ...
echo $hello      # 375 ##  This *is* a variable reference, i.e. shows the value.
echo ${hello}    # 375 ##  Likewise a variable reference, as above.
```

assignment disambiguation with {}

In the previous slide: “In contexts where the $variable syntax causes an error, the longer form ${variable} may work”. This is called variable disambiguation.

Example:
If the variable $type contains a singular noun and we want to transform it on a plural one adding an ‘s’, we can't simply add an ‘s’ character to $type since that would turn it into a different variable, $types.
Although we could utilize code contortions such as
echo "Found 42 "$type"s"

the best way to solve this problem is to use curly braces:
echo "Found 42 ${type}s",
which allows us to tell bash where the name of a variable starts and ends

Exercise: bash variables

Try:

1) STR=Hello World!
   STR=”Hello World!”
   echo $STR

2) Try assignment and echo the variable content:

   a=5324

   a=(1 3 4 6 5 otto) # array

3) Very simple backup script example:
   OF=/tmp/my-backup-$(date +%Y%m%d).tgz
      tar -czf $OF ./subdir_of_where_i_am

bash variables: assignment examples

“naked variable”, i.e. lacking ‘$’ in front, is when a variable is being assigned, rather than referenced.

```
# Assignment simple
a=879 ;  echo "The value of \"a\" is $a."

# Assignment using 'let' (arithmetic expression)
let a=16+5;  echo "The value of \"a\" is now $a."

# In a 'for' loop (see for details later in this lesson):
echo -n "Values of \"a\" in the loop are: "
for a in 7 8 9 11
do
  echo -n "$a "
done

# In a 'read' statement (also a type of assignment):
echo -n "Enter \"a\" "
read a
echo "The value of \"a\" is now $a."
```

bash variables: assignment examples(2)

```
#!/bin/bash
# With command substitution

a=$(echo Hello!)   # Assigns result of 'echo' command to 'a' ...
echo $a

a=$(ls -l)         # Assigns result of 'ls -l' command to 'a'
echo $a            # Unquoted, however, it removes tabs and newlines.

echo "$a"          # The quoted variable preserves whitespace.
```

Exercise 3: practice with variables assignment

Try different variable assignments and print the variable content to standard output

- Simple assignment
- Command output assignment

bash variables: quoting

Quoting means just that, bracketing a string in quotes.
This has the effect of protecting special characters in the string from reinterpretation or expansion by the shell or shell script. (A character is "special" if it has an interpretation other than its literal meaning. For example, the asterisk * represents a wild card character in Regular Expressions).

Partial quoting consists in enclosing a referenced value in double quotes (" ... "). This does not interfere with variable substitution. Sometimes referred also as "weak quoting."

Full quoting consists in using single quotes ('...').
It causes the variable name to be used literally, and no substitution will take place.

Examples (Try):

```
a=352
echo $a    # 352
echo “$a”  # 352
echo ‘$a’  # $a
```

=> Quoting a variable preserves whitespaces.

Exercise 2: variables assignment and quoting

In a bash script:

- Assign a variable
- Print the variable value
- Print a string containing the variable value
- Print a string containing the partial quoted variable
- Print the same string fully quoted
- Assign a variable containing multiple spaces
- Print this new variable
- Print this new variable quoted

- Run the script
- Run the script redirecting the output on a file

Bash Arithmetic Expansion (1)

★Arithmetic expansion provides a powerful tool for performing (integer) arithmetic operations.
Translating a string into a numerical expression is relatively straightforward using `expr`, backticks, double parentheses, or let.

★ expr examples:

```
z=15
z=$(expr $z + 3)
echo $z
```

Demonstrating some of the uses of 'expr'

```
# Arithmetic Operators
a=$(expr 5 + 3)
echo "5 + 3 = $a"

a=$(expr $a + 1)
# incrementing a variable
echo "a + 1 = $a"

# modulo
a=$(expr 5 % 3)
echo "5 mod 3 = $a"
```

Bash Arithmetic Expansion (2)

★Parentheses examples:

$((EXPRESSION)) is arithmetic expansion.

Not to be confused with + command substitution.

Examples:

```
n=0
echo "n = $n"                    # n = 0
(( n += 1 ))                     # Increment.
echo "n = $n"                    # n = 1
```

Bash Arithmetic Expansion (3)

★`let` let does exactly what (( )) do.

Examples:

```
z=0
let z=z+3
let "z += 3"  #  Quotes permit the use of spaces in
              #  variable assignment.

              #  The 'let' operator actually performs
              #  arithmetic evaluation,
              #  rather than expansion.
              #  [try: echo $(let "z += 7")]
echo $z
```

Array in bash

★Initialize an array: arrays in Bash can contain both numbers and strings:

- Initialization with all elements of the same type (numbers)

```
myArray=(1 2 4 8 16 32 64 128)
```

- Initialization with mixed types elements

```
myArray=(1 2 "three" 4 "five")
```

★Make sure to leave no spaces around the equal sign. Otherwise, Bash will treat the variable name as a program to execute, and the = as its first parameter!

Retrieve array elements in bash

★Although Bash variables don't generally require curly brackets, they are required for arrays.

In turn, this allows us to specify the index to access:

echo ${myArray[1]} returns the second element of the array (indexes starts from zero).

★Not including brackets

echo $allThreads[1] leads Bash to treat [1] as a string and output it as such.

Some useful array operations

Syntax	Result
arr=()	Create an empty array
arr=(1 2 3)	Initialize array
${arr[2]}	Retrieve third element
${arr[@]}	Retrieve all elements
${!arr[@]}	Retrieve array indices
${#arr[@]}	Calculate array size
arr[0]=3	Overwrite 1st element
arr+=(4)	Append value(s)
str=$(ls)	Save ls output as a string
arr=($(ls))	Save ls output as an array of files
${arr[@]:s:n}	Retrieve n elements starting at index s

Exercise: retrieve array elements

★Initialize three arrays:

- One with only numbers
- One with only strings
- One with mixed elements

★ Retrieve the first, and the third element of each one

★ Write a script that:

- enter the home directory (hint. HOME environmental variable);
- save all the files in the home directory as bash array;
- print the last element of the array