

# Handling data files

## Writing bash scripts

Florido Paganelli

Lund University

`florido.paganelli@hep.lu.se`

Fysikum, Hus A, Room 403

Support:

- send me an email or use Canvas
- personal Zoom room:

<https://lu-se.zoom.us/j/2485752983>

# Outline

- Goals
- Motivation:
  - Datasets
  - Automation using scripting
- Introduction to scripting
- Bash
  - Introduction
  - Tutorial part 1: basic concepts (variables, exit values, conditions)
  - Tutorial part 2: advanced concepts (environment, loops)
  - Useful commands

# Goals and non-goals of this tutorial

- Goals:
  - Understand the concept of **dataset format**
  - Being able to write a **bash script**.
  - Understanding the concepts of **Variable, Environment, binding, scope**.
- Non-goal:
  - Become a script-fu master. It takes long time for the black belt :)
  - Become a coder. We cannot do this in a lecture, there's plenty of dedicated courses out there

# Motivation 1: Handling datasets

Excerpt from the Datasets document on  
Canvas

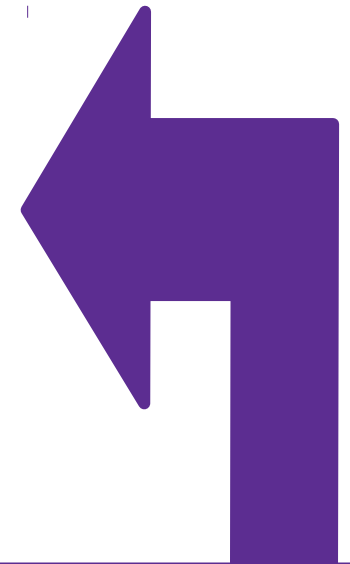
# Typical scientist workflow summary

1. Understand datasets **format**

2. **Cleanup** data using tools like bash commands, python, perl...

3. Write code to **process** data in languages like C, C++

4. Write scripts in Bash, Python, Perl  
To **automate** steps 2 and 3 on multiple datasets



# Typical scientist workflow

- **Someone** (usually your supervisor) gives you reference to some data and some obscure code written by elders who now moved to the end of the known universe
  - Nobody knows what the **data** looks like and what it contains – just that is it about your science!
  - Nobody has any idea what the **code** is like and how to change it. No documentation, no one left alive to tell you
- **You** have to figure out most of the the details by yourself
- Few tips in separate slides

# Datasets

- A dataset is some digital collection, maybe a file or a set of files, that **contains data** we want to use. Here are the typical traits of a dataset:
- **Scope:** The knowledge area it targets. Examples: archeology, budget, weather forecast...
- **Format**
  - A *format* is a **set of rules** that define in a rigorous manner how the content of the dataset should be read and written, what are their meanings and the relationship among the dataset information
  - The format can be a well know data format, more or less standardized, or some custom data format created by some community.
  - A **description** of the format is usually provided by the community that generated the dataset. It is very rare that a dataset contains information about its format.
  - Very common format **names**
    - **CSV** (Comma **S**eparated **V**alues)
    - **XML** (e**X**perimental **M**arkup **L**anguage)
    - **JSON** (Java**S**cript **O**bject **N**otation)
  - Sometimes the format can be made explicit in the **file extension**, but it is not necessary: `fridaythe13.json`
- **Meaning of each data entry:** what is the data **exactly** about.
  - In a budget, it's probably whatever concerned an expense, the goods bought, how they were bought, maybe billing numbers...

# Datasets can be “dirty” and need to be “cleaned”

- The data is not always as you expect. Close inspection might reveal inconsistencies and corner cases that have to be “*sanitized*” or “*validated*”, or simply you need a subset of the whole dataset.
- In most cases you will need to *rework the dataset* in order to process it with your code
  - In any case, **never tamper the original dataset**. Do all the changes on a **separate copy**.
- Devil is in the detail:
  - **Format consistency check**: is it consistent or some data entries do not respect the format rules?
  - **Meaning consistency check**: is the data meaningful or totally nonsense, for example a temperature of 2000 Celsius in Kiruna in January do not make sense... maybe the sensor had an issue!
  - **file format issues**: Look out for **encodings, invisible or non-ASCII characters**. These are usually symbols for non English-US languages, or **control characters**
- Bash and the bash commands are usually good for cleaning **text files datasets**.



# Example from project

filename: smhi-opendata\_1\_53260\_20210906\_214756.csv

```
Stationsnamn;Klimatnummer;Mäthöjd (meter över marken)
Ystad;53260;2.0

Parameternamn;Beskrivning;Enhet
Lufttemperatur;momentanvärde, 1 gång/tim;degree celsius

Tidsperiod (fr.o.m);Tidsperiod (t.o.m);Höjd (meter över havet);Latitud (decimalgrader);Longitud (decimalgrader)
1949-01-01 00:00:00;1963-09-16 23:59:59;13.0;55.4410;13.8278
1963-09-17 00:00:00;1983-08-31 23:59:59;32.0;55.4410;13.8278

Datum;Tid (UTC);Lufttemperatur;Kvalitet;;Tidsutsnitt:
1949-01-01;00:00:00;3.5;Y;;Kvalitetskontrollerade historiska data (utom de senaste 3 mån)
1949-01-01;06:00:00;3.5;Y;;Tidsperiod (fr.o.m.) = 1949-01-01 00:00:00 (UTC)
1949-01-01;12:00:00;4.2;Y;;Tidsperiod (t.o.m.) = 1983-08-31 23:59:59 (UTC)
1949-01-01;18:00:00;5.5;Y;;Samplingstid = Ej angivet
1949-01-02;00:00:00;4.3;Y;;
1949-01-02;06:00:00;4.6;Y;;Kvalitetskoderna:
1949-01-02;12:00:00;6.2;Y;;Grön (G) = Kontrollerade och godkända värden.
1949-01-02;18:00:00;5.0;Y;;Gul (Y) = Misstänkta eller aggregerade värden. Grovt kontrollerade
1949-01-03;00:00:00;4.2;Y;;
1949-01-03;06:00:00;4.0;Y;;Orsaker till saknade data:
1949-01-03;12:00:00;4.6;Y;; stationen eller givaren har varit ur funktion.
1949-01-03;18:00:00;5.0;Y
1949-01-04;00:00:00;3.2;Y
1949-01-04;06:00:00;1.6;Y
1949-01-04;12:00:00;3.0;Y
1949-01-04;18:00:00;2.0;Y
1949-01-05;00:00:00;1.5;Y
1949-01-05;06:00:00;-0.8;Y
```

# Motivation 2: Automation

# Workflows with various tools

- In the free software/open source community everyone shares knowledge about coding.
  - This usually means that someone's work is based on someone else's
  - This generated a style of creating software that is a mix of different programming languages, tools, practices:  
**composing different applications to achieve a goal**
- It is very common that your C++ code will require some **preparation** before the build for which leads to **tedious repetitive** commands to type in
- **“tedious repetitive”** is what a computer is good at.  
**Tip: use a computer to do tedious and repetitive stuff**  
A human has better things to do in life than monkey-coding!
- *Scripting languages like bash* are very good to automate boring work.

# Automation and composition of languages

- Cornerstone of open source programming: if something exist that does a task, and it does it good, use it and do not rewrite code
- **Automation** of repetitive tasks
- Make use of interoperability within languages
- Technique: identify subproblems and separate tasks, increasing “debuggability”
- **Choose the right command/language for each subtask, compose it with bash**

# A bunch of commands you should know about

The “GNU userland”, the collection of commands that are usually shipped with linux, it’s a great collection of command line tools that can do a lot for you. Here I write some that are notable, with links to examples. They mostly do string operations and can be used to cleanup or reformat a dataset.

- **grep** - find all the occurrences of a substring inside a file.  
Example: `grep expression to find filename.txt`  
<https://www.geeksforgeeks.org/grep-command-in-unixlinux/>
- **sed** - substitute strings. The most used form is  
`sed 's/pattern to find/pattern to substitute/' filename.txt`  
<https://www.geeksforgeeks.org/sed-command-in-linux-unix-with-examples/>
- **cat** - print a file  
`cat filename.txt`
- **head, tail** - print n lines from the top/bottom of a file, see Tutorial 2 slides  
`head -10 filename.txt ; tail -10 filename.txt`
- **cut** - remove section from each line of a file - can be used to extract columns  
`cut -d, -f5 filename.txt`  
<https://www.thegeekstuff.com/2013/06/cut-command-examples/>
- **tr** - translate (substitute) characters  
`cat /etc/services | tr s z` (will make every s -> z)  
<https://www.thegeekstuff.com/2012/12/linux-tr-command>
- **awk** - a powerful line editor that can be programmed for tasks  
<https://likegeeks.com/awk-command/>
- **curl** and **wget** - programs used to download files  
<https://www.keycdn.com/support/popular-curl-examples>  
<http://www.linuxandubuntu.com/home/12-practical-examples-of-wget-command-on-linux>
- **sort** - orders lines of a file give a certain criteria, eventually based on columns or fields  
`sort -r -k2 -h /etc/services`  
<https://www.geeksforgeeks.org/sort-command-linuxunix-examples/>
- **wc** - bytes, chars and lines counter  
`wc -l /etc/services`  
<https://www.tecmint.com/wc-command-examples/>

# A case study for tutorial

- We want to go from the file format on the left side to the one on the right side.

```
Stationsnamn;Klimatnummer;Mäthöjd (meter över marken)
Ystad;53260;2.0
```

```
Parameternamn;Beskrivning;Enhet
Lufttemperatur;momentanvärde, 1 gång/tim;degree celsius
```

```
Tidsperiod (fr.o.m.);Tidsperiod (t.o.m.);Höjd (meter över havet);Latitud (decimalgrader);Longitud (decimalgrader)
1949-01-01 00:00:00;1963-09-16 23:59:59;13.0;55.4410;13.8278
1963-09-17 00:00:00;1983-08-31 23:59:59;32.0;55.4410;13.8278
```

```
Datum;Tid (UTC);Lufttemperatur;Kvalitet;;Tidsutsnitt:
1949-01-01;00:00:00;3.5;Y;;Kvalitetskontrollerade historiska data (utom de senaste 3 månaderna)
1949-01-01;06:00:00;3.5;Y;;Tidsperiod (fr.o.m.) = 1949-01-01 00:00:00 (UTC)
1949-01-01;12:00:00;4.2;Y;;Tidsperiod (t.o.m.) = 1983-08-31 23:59:59 (UTC)
1949-01-01;18:00:00;5.5;Y;;Samplingstid = Ej angivet
1949-01-02;00:00:00;4.3;Y;;
1949-01-02;06:00:00;4.6;Y;;Kvalitetskoderna:
1949-01-02;12:00:00;6.2;Y;;Grön (G) = Kontrollerade och godkända värden.
1949-01-02;18:00:00;5.0;Y;;Gul (Y) = Misstänkta eller aggregerade värden. Grovt kontrollerade
1949-01-03;00:00:00;4.2;Y;;
1949-01-03;06:00:00;4.0;Y;;Orsaker till saknade data:
1949-01-03;12:00:00;4.6;Y;; stationen eller givaren har varit ur funktion.
1949-01-03;18:00:00;5.0;Y
1949-01-04;00:00:00;3.2;Y
1949-01-04;06:00:00;1.6;Y
1949-01-04;12:00:00;3.0;Y
1949-01-04;18:00:00;2.0;Y
1949-01-05;00:00:00;1.5;Y
1949-01-05;06:00:00;-0.8;Y
```

```
1949-01-01 00:00:00 3.5 Y
1949-01-01 06:00:00 3.5 Y
1949-01-01 12:00:00 4.2 Y
1949-01-01 18:00:00 5.5 Y
1949-01-02 00:00:00 4.3 Y
1949-01-02 06:00:00 4.6 Y
1949-01-02 12:00:00 6.2 Y
1949-01-02 18:00:00 5.0 Y
1949-01-03 00:00:00 4.2 Y
1949-01-03 06:00:00 4.0 Y
1949-01-03 12:00:00 4.6 Y
1949-01-03 18:00:00 5.0 Y
1949-01-04 00:00:00 3.2 Y
1949-01-04 06:00:00 1.6 Y
1949-01-04 12:00:00 3.0 Y
1949-01-04 18:00:00 2.0 Y
1949-01-05 00:00:00 1.5 Y
1949-01-05 06:00:00 -0.8 Y
```

## Simplified raw data

- No headers, just raw data
- Removal of unused fields
- spaces instead of semicolon

## Not very easy to process:

- Metadata headers
- inconsistent field structure, some fields are used for comments
- semicolon may not be nice for C++

# What coder are you?

- Over the years I understood there are two kinds of coders:
  - task-oriented
    - They need a task to be motivated to learn the details of a programming language
  - knowledge-oriented
    - They need basic knowledge of the basic of a language before they dive into coding with that language

# Two ways to do the tutorial

- I will provide
  - A set of exercises explaining the basics of BASH,
  - A pseudocode file and my solution to the case study.
- Your task is to **try to write code in the pseudocode file** and solve the case study.
- Depending on which coder are you, you can complete the Tutorial in two ways. Or mix.
  - Task-oriented: focus on solving the case study. Read the pseudocode and follow the suggestions on what exercises to do in the tutorial that help you writing your own code
  - Knowledge-oriented: focus on concepts and features of the language. Do all the exercises in the tutorial until the end, then look at the pseudocode and try to write your own code.
- In both cases you can always look at the solution to see how I did it.
  - Maybe you come up with smarter ideas, my code is just one way of solving the problem.



# Check previous years homework

Check the solutions of previous year assignments on github or the course webpage:

- <https://github.com/floridop/MNXB01-2021/tree/main/floridopag/tutorial3/homework3>
- <https://github.com/floridop/MNXB01-2020/tree/master/floridopag/tutorial3/homework3>
- <https://github.com/floridop/MNXB01-2019/tree/master/floridopag/tutorial3/homework3>
- <http://www.hep.lu.se/courses/MNXB01/index-2018.html>
  - <https://github.com/floridop/MNXB01-2018/tree/master/floridopag/HW3>
- <http://www.hep.lu.se/courses/MNXB01/index-2017.html>
  - <https://github.com/floridop/MNXB01-2017/tree/master/flopaganelli/HW3b>
- <http://www.hep.lu.se/courses/MNXB01/index-2016.html>
- <http://www.hep.lu.se/courses/MNXB01/index-2015.html>

# Not published yet: Homework3

- It's about writing a bash script
- Not yet completed, will be described on canvas
- It may involve reusing the `smhicleaner.sh` script from the case study

# Introduction to BASH

# Scripting vs coding

- The word script is taken from a theatrical play script: a description of the environment on stage, a sequence of lines and gestures to do
- There is no practical difference between writing code in a compiled language and a scripted one.
- The main *runtime* difference is that scripted languages **do not require compilation**.
  - In particular, Bash commands are already precompiled, and you can use bash to execute existing compiled programs.




# BASH

- Bash stands for **B**ourne-**A**gain **S**hell. It's a rewrite by the GNU member Brian Fox of one of the unix `sh` shells, called Bourne shell by its author surname (Stephen Bourne).
- **Command Interpreter**: defines a language to automate and manipulate the **output** and the **execution** of commands.
  - Some commands are part of the language, the so-called “built-ins”
  - Many of the things one can do actually depends on commands installed on a machine independently from bash, like most applications, so it is not possible to be extensive in explaining it.
- Yesterday you learned a few existing commands in bash and applications, today we will write programs that use them.

# Mastering bash

- To be able to write good programs in bash one needs two things:
  - 1) Have a good understanding of the bash syntax and features
    - Today's tutorial!
  - 2) Have a good knowledge of the commands that one can run in bash
- commands: the GNU programs and other third party tools. They are best learned when one has a specific problem to solve, but in the end you actually learn by experience and internet search. Some we saw yesterday, some suggestions with hints to where to search information will be in the case study.
- Use the section 5.3 in the manual and slide 13!

# Notation

- There's a set of symbols and idioms that are commonly used in command line tutorials and you should know about. The description of the grammar of a command is often called **synopsis**, or brief summary.
- **Spacing.** In general there is **always** a space between a command and every one of its options, that is, every word of a command that is shown in these slides.  
However, in some cases it might be tricky to see it, and I will use the symbol . For example  
- **command**  
This graphics above is meant to represent a **command**. You are supposed to write exactly as it looks.
- **command <argument>**  
The **<>** (**angle brackets**) are used to identify a **mandatory** argument of the command. The command will NOT work without the things in the angle bracket.  
The above usually means to run the command and to **substitute** the string <argument> with the argument **without angle brackets**.  
Remember, in most languages brackets have a special meaning. The special meaning of the angle brackets was shown in the CLI tutorial.
- **command ARGUMENT**  
In **man** pages, sometimes **capital letters** are used instead of the angle brackets <>. The meaning is exactly the same as the angle brackets, the capitalized string means **mandatory**. **We will not use this notation in this tutorial** because it might be confusing, but you will find it in the linux **man** pages
- **command <argument> [<argument>]**  
The **[ ]** (**square brackets**) are used to identify and **optional** part of the command. The command will work if you omit the content of the square brackets [].  
However, if you add a second argument, it must be as defined within the angle brackets <>.
- **command [<argument1> | <argument2>]**  
In command descriptions, the **|** (**pipe symbol**) is used to identify a **mutually exclusive** part of the command. You can use **EITHER** <argument1> **OR** <argument2> but **NOT both of them**.  
This is inherited from formal grammar notations.  
In code snippets or pieces of code, the pipe is part of the code and must be copied/written as it is.

# A bash script and its components

- **Bash** is not really a programming language. It is more like a **command scripting language** for automation of tasks, with some programming language features.
- Instead of libraries you will mainly use the **GNU/Linux userland** and **GNU/Linux coreutils** software, a set of commands that help automate common tasks, or other bash scripts.
  - However, one can write “bash libraries” as scripts that define usable functions and *source* these scripts in another script (see slide 87)
- A **bash script** is nothing more than a sequence of commands written in a file.
- The bash interpreter will process those in sequence, from the top line to the bottom
- Like C++, it is possible to define **variables** and **control structures** in the scripting language.
- However, the bash script language has little to share with the complexity of C++. All that it can do is to **execute commands, test conditions, and store things in variables**.
- Most commands we will see today are documented on man. You can type `man bash` to read the full documentation.
- Consider the following code, a script called `getcpuinfo.sh`:

```
#!/bin/bash

# 1. use the cat command to print the file /proc/cpuinfo,
#    a system file that contains information about the cpu
# 2. extract the first two lines of the above output with head
# 3. store the output of head in the CPUINFO variable
# it is all done in the following one line!
CPUINFO=$(cat /proc/cpuinfo | head -2)

# write the content of CPUINFO to screen
echo "$CPUINFO"
```

Execution result:

```
$ ./getcpuinfo.sh
Processor: 0
vendor_id: AuthenticAMD
```



# Anatomy of a bash script

`#!/bin/bash`

The first line has a special syntax: `#!` tells bash which **interpreter** to use. It might be another shell!

`# put the output of cat in the variable CPUINFO`

Every other line starting with a hash `#` is a **comment**. The interpreter ignores everything that follows until the end of line. Useful to describe code to human readers.

## commands

`CPUINFO=`

`$( cat /proc/cpuinfo | head -7 )`

This tells bash to **execute** a command and return its output.

A **variable definition** is any string followed by a `=` symbol. It is a convention to use capital letters. Remember that *case matters*, `cpuinfo` is different from `CPUINFO`!

`# write the content of CPUINFO to screen`

`echo "$CPUINFO"`

A **variable call** is any **variable name** prefixed by the `$` symbol. Case does matter here. The quotes affect the output, that in this case depends on how the `echo` command works. The `$` symbol stands for “**give me the value contained in that variable**”

# Executing a script

- The script can be **made executable** as if it were a command.

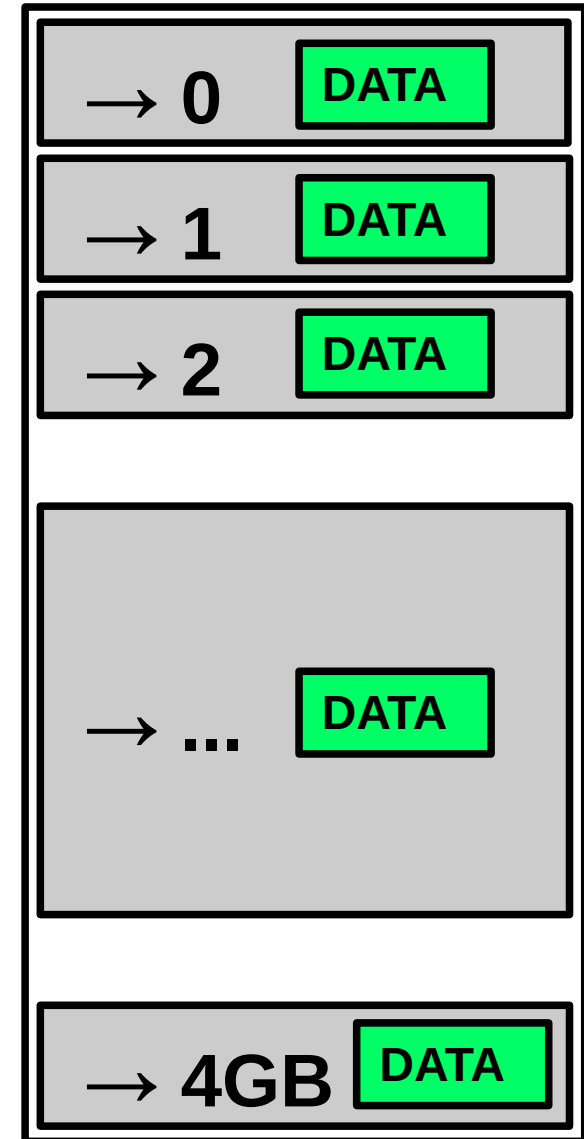
```
pflorido@tjatte:~> chmod +x getcpuinfo.sh
```

- If you forgot to make the file executable, you will get the following error:  
bash: ./getcpuinfo.sh: Permission denied
- To **run** or **execute** a file in the current directory, prefix it with **./**

```
pflorido@tjatte:~> ./getcpuinfo.sh
processor : 0
vendor_id : GenuineIntel
cpu family : 6
model      : 15
model name : Intel(R) Core(TM)2 CPU          6400 @ 2.13GHz
stepping   : 6
cpu MHz    : 2127.650
```

# Digression: Addressing memory (RAM)

- Computer memory is divided in a certain number of **locations**.
- A physical memory element at a specific location is like a register, and has a size in bit. Usually is 8 bits, a byte.
- A location is a memory space identified by a **memory address**
- A memory address is an integer **number**.
- This number is usually called **pointer** ( → ), as it points to a memory location.

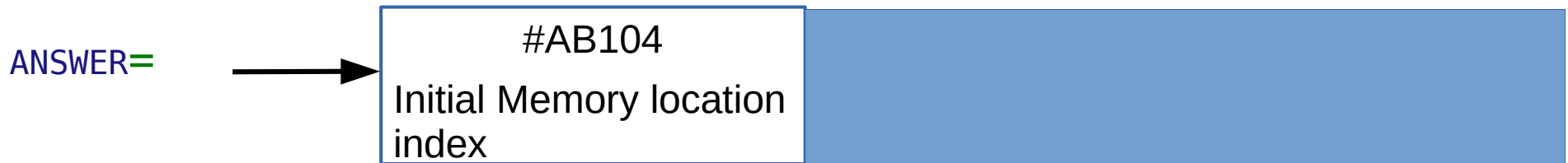


# Variables: definition, initialization

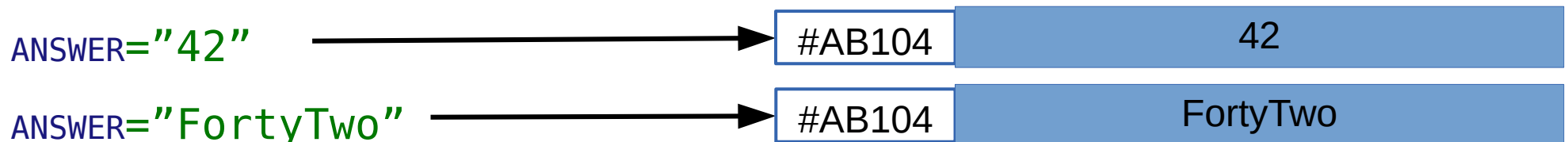
- A **variable** is an identifier, a name, for a memory location.
- To **define** a variable means to tell the **interpreter** to find a free memory space for that variable. This memory space has an index.
- In bash, you define a variable by simply writing a string with CAPITAL LETTERS (by convention) that starts with a letter **on the left side** of the symbol =  
Note: BASH doesn't like spaces that much!

ANSWER=

- If a bash variable is **not initialized**, the memory space at that index contains the blank/empty string



- To **initialize** a variable is to **assign** a **value** to it.  
It means putting such value inside the memory location identified by that variable name.
- In bash this is done by writing a value on the **right side** of the equal sign =



# Variable types in bash

- In BASH, variables have *no explicitly defined type*, because actually there is **only one type**.
- It is **implicitly assumed** that the content is a **string: a sequence of characters**.  
The maximum size depends on the system.
  - Memory Allocation is always done dynamically depending on the assigned value.
  - **You will never see the memory pointer in bash.**
  - **Consequence:** Doing arithmetic with bash is a **bad idea**. Bash does not understand numbers so well...
    - There are, however, a special syntax and operators that force basic arithmetic interpretation (integer sum, logical algebra)

Var name	Var type	Associated size	Initial tentative logical memory location pointer	value
ANSWER	Always string	Depends on system configuration	#AB104	42 as a string
ANSWER	Always string	Depends on system configuration	#AB104	FortyTwo

# Variables: retrieving values

- So far we've seen how to assign a value to a variable. But how to *read* or *retrieve* such value from the computer's memory?
- In bash one simply prefixes the variable with the \$ (dollar) sign.
- **\$ANSWER** returns the value of the **ANSWER** variable. This is sometimes also named **(re)calling a variable value**

# Calling variables values in different ways

- `$VAR` returns the value contained in the variable called `VAR`.
- `${VAR}` also returns the value contained in the variable called `VAR` but it allows you to manipulate the contents of the variable. This is called *shell parameter expansion*. In this course we will not see it in detail, but we will use it just because it makes it **easier to spot the boundaries of the variable name**. It can be used to concatenate string values and strings, like:  

```
${TARGETDIR}/*;
```

someone who reads the code can clearly see that the name of the variable is `TARGETDIR` and that `/*` are something else.
- For the curious, more about *shell parameter expansion* here:  
[https://www.gnu.org/software/bash/manual/html\\_node/Shell-Parameter-Expansion.html](https://www.gnu.org/software/bash/manual/html_node/Shell-Parameter-Expansion.html)

# Best practices coding in bash

- Get used to reason in terms of **command outputs** and **command inputs**, it is the real thing that bash can do. For more complicated things you may need a real programming language.
- **Test** a command **in the command line** first, then add it to a script.
- Use the **debug mode** when you do not understand what is going on (see slide 41)
  - `echo` (print out) the content of a variable **on screen** to see what it contains
- Test you script on different paths/filenames/inputs
- If in your computer, **never run a script you're testing as superuser/root**



# Bash Tutorial part 1

- Exercises and their solutions
- Create and execute a script
- Variables
- Debugging
- Passing input parameters to a script
- The exit status and exit command
- Taking decisions: conditions and IF statement
- Command substitution and pipes

# copy the bash examples/exercises to your home

Do not copy paste the commands below. Please type them.  
Copy pasting from PDF doesn't always work properly.

- Change Directory into your home.

```
cd ~
```

- Recursively copy the tutorial3 directory to your home folder from our shared folder:

```
cp -ar /projects/hep/fs10/mnxb01/tutorial3 ~
```

- Enter the tutorial folder you just copied:

- To see Examples:

- `cd ~/tutorial3/examples/bash`

- To see exercises:

- `cd ~/tutorial3/exercises/`

- To see the case study:

- `cd ~/tutorial3/casestudy/`

- Remember: you can list the folder contents to see the scripts with

- `ls -l`

# Exercises and their solutions

1) cd into the exercise folder ex3.x  
where x is the number of the  
exercise

Example, for exercise 3.1:

```
cd ~/tutorial3/exercises/ex3.2
```

2) perform the requested tasks described in  
these slides you're reading

3) Check the expected output  
(cat output)

4) Try to write the code or run the script  
so that it outputs the same

5) If you do not manage, check the solution in solution/

- For exercises that require running commands, the solution can be inspected with  
cat command\_to\_run

You may want to open multiple terminals, one to check the solution,  
one to run the command(s).

- Folder structure:

- tutorial3/
  - exercises/
    - ex3.x/
      - output
      - solution/
        - command\_to\_run
        - example.sh

# How to start working on each exercise

1) Access the exercise directory

Example: exercise 3.1

```
cd ~/tutorial3/exercises/ex3.1
```

2) Open Pluma

```
pluma &
```

- Read about how to save files in the MNXB01-manual.pdf

3) Read the exercise description in these slides  
(there is no exercise description in the folders!)

4) Perform the requested tasks

# How to start working on the case study

- Read the README.md file in the folder `casestudy` to understand more about the case study
- You can see it nicely formatted on my github, maybe easier to read:
- <https://github.com/floridop/MNXB01-2021/blob/main/floridopag/tutorial3/casestudy/README.md>

# Create a script

- Read slides 22-24 if you are confused about the syntax of these exercises.
- **Exercise 3.1** (write a bash script): Open Pluma (see MNXB01-manual.pdf) write and save the following code as file `answers.sh`

```
#!/bin/bash  
  
# define and initialize the ANSWER variable  
ANSWER=42  
  
# print the content of ANSWER to screen  
echo "$ANSWER"
```

- **Exercise 3.2** (execute a bash script): make the script `answers.sh` executable and execute it as described in slide 23.
- **Exercise 3.3** (echo): Familiarize with the `echo` command. It is used to print out information to the screen.  
Edit `answers.sh` so that at the end of the program it prints out “The content of the variable ANSWER is 42”
- **Exercise 3.4** (modify scripts): Modify the content of ANSWER with `42+42`, save and execute again. Is it what you expected?  
Everything is a string (sequence of characters) in bash, by default there is no numerical calculations.

# Predefined variables in scripts

- **Prefixed by the \$ symbol**, they are instantiated automatically in bash at the start of the shell program.  
(They are actually automatically *sourced in the environment*, we'll talk about this later)
- **Various:**
  - **\$PATH**: list of paths where executable commands are
  - **\$PS1**: prompt format
  - **\$SHELLOPTS**: options with which the shell is run
  - **\$UID**: User ID of the user running the script
- **Process info and status codes:**
  - **\$\$**: process identifier (PID) of the script itself.  
The **PID** is an **integer number** that the operating systems assigns to a binary file once it is ran, that is, when it becomes a process. **It uniquely identifies a running program** until the machine is shut down. See Balazs slides for Tutorial 2 and the Advanced Topics in the Lecture 3 module on canvas.
  - **\$?**: exit code of the last executed command (0 if it ended without errors, any other number otherwise).  
More about it later in the tutorial.
  - **\$\_**: PID of last command executed in background
- **Script parameters/arguments**: **\$#, \$0, \$1, \$2....**
  - **\$#** is the number of arguments passed to the script
  - **\$0** is the name of the script itself as called to be executed
  - **\$1 . . n** is each string that follows the name of the script.
  - **\$\*** is all the parameters on a line.

# Exercises

## Exercise 3.5:

### What is the predefined **PATH** variable?

During Balazs' lectures we ran commands without typing a `./` in front of them. The reason is: the system has a list of paths where to find these commands.

This list is contained in the predefined variable **PATH**.

Add the following line at the end of the `answers.sh`:

```
echo "PATH value is $PATH"
```

Save and execute the script again: this line above will show the folder path where the the system looks for executables.

Usually the directory where one creates a script is not in the `$PATH` variable. For all executables that are not in `PATH`, one has to add `./` in front for the system to find them.

It means: "the script I want to run is in the current directory, do not search in `PATH` for it!"



# Exercises

## Exercise 3.6 (enable/disable debugging mode):

Enable **Debugging** to debug the script, that is, see what is doing while running, modify the *first line* of `answers.sh` as below, by adding a `-x` option:

```
#!/bin/bash -x
```

Save the file and execute it again. See the differences in the output.

- The lines starting with `+` show what line the interpreter is **processing**

- the lines **without** `+` are the output **result** of the process.

```
> ./answers.sh
```

Processing variable: store 42 inside ANSWER

```
+ ANSWER=42
```

processing echo command

```
+ echo 42
```

```
42
```

Result of echo command execution: print 42 on screen

```
+ echo "PATH value is /nfs/users/floridop/bin:/usr/l  
"PATH value is /nfs/users/floridop/bin:/usr/local/sb
```

You may delete the `'-x'` after this exercise, and just add it back when you do not understand what the code is doing.

# Using parameters and quotes

- **Exercise 3.7:** Let's modify the `answers.sh` script to take as input the number it has to print. Using the predefined variable `$1` in slide 35:

```
#!/bin/bash

# set the variable to the first input parameter
# to the answers.sh script
ANSWER=$1

# print the content of ANSWER to screen
echo "You asked me to print: $ANSWER"
```

- **Exercise 3.8:** execute `answers.sh` passing a value of your choice, for example:  
`./answers.sh FortyTwo`
- **Exercise 3.9:** Pass the string `(42)` (including the parentheses). What happens?
  - Certain characters are special in Bash (see Tutorial 2). If you want to pass them as string, you must enclose them in quotes `'` or double quotes `"`.  
Try again with the following:
    - `"(42)"`
    - `'(42)'`
    - `"$PATH"`
    - `'$PATH'`
- The meaning of the quotes is different:
  - The single quote `'` is *verbatim*, that is, what is inside the quotes is taken exactly as it is.
  - The double quote `"` allows to resolve/fetch the value of variables, as in `echo`

# Predefined variables example

```
1. #!/bin/bash
2. # predefinedvars.sh
3. # call with: ./predefinedvars.sh arg1 arg2 arg3
4. #
5.
6. # print out info about arguments to this script
7. echo "Number of arguments: $#"
```

8. echo "Name of this script: \$0"

9. echo "Arguments: \$1 \$2 \$3 \$4"

10.

11. # print this script's Process IDentifier:

12. echo "PID is \$\$"

Let's consider the predefined variables in slide 35 and the script called `predefinedvars.sh` in folder `ex3.10`.

**Exercise 3.10:** Run the script by passing five arguments:

```
./predefinedvars.sh arg1 arg2 arg3 arg4 arg5
```

Note how line 9 cannot print the fifth argument since I didn't list \$5.

**Exercise 3.11:** modify the script so to check the output of at least the two predefined variables `$*` and `$@`

Hint: you can *escape* special characters with a backslash in order to print them so that bash does not *interpret them as variables*:

- `"\ $*"` will print `$*`
- quote the var with `' $* '`

# The process exit status variable \$?

- Every process in an operating system has an **exit status**.
- It is an **integer** representing the status of an executed program after it terminated.
- By convention:
  - **0**: Program completed **without errors**
  - **anything else**: Program **failed with errors**
- When a program completes, bash automatically saves its status in the `?` variable. It's status of the "last executed command"
- ▶ We read this variable to **check** if the commands in our scripts are doing as expected.

Example:

I can create a file in my home folder `~`, and the exit status is 0 (file created)

If I try to create it in the root filesystem where I have no permission, I get an error.  
Exit status 1

```
[pflorido@pptest-iridium ~]$ touch ~/myfile
[pflorido@pptest-iridium ~]$ echo $?
0
[pflorido@pptest-iridium ~]$ touch /myfile
touch: cannot touch `/myfile': Permission denied
[pflorido@pptest-iridium ~]$ echo $?
1
```

# Use of `$?`, process exit status

- When you write a script, you should always check what is the exit status of the last command especially if this prevents the logic of your script to continue.  
**IMPORTANT: BASH will not stop if any of the commands in your script fail. It will show the error and continue executing the script.**
  - You can *check the exit value* by getting the value of the `$?` variable as in the example:  
`echo $?`
  - You can also store it in a variable for future use:  
`touch ~/myfile`  
`MYERRORSTATUS=$?`
  - REMEMBER: `$?` just shows the return value of the **most recent executed command**. It is therefore essential that you run any check right after a commands has been executed in your script.
- **Exercise 3.12:** Run the following commands and check their exit status.
  - `ls /`
  - `ls /etc/MNXB01`
  - `ls /; ls /etc/MNXB01`
  - `ls /etc/MNXB01; ls /`
- Notice how the use of `;` that is used to execute a list of commands affects the result. `$?` Refers only to the last executed command and not to the whole list.

# The `exit` command

- If the return status of one of the commands in your scripts terminates with error, the `exit` command can be used to terminate the program exactly where `exit` is called, that is, to break cycles and exit the program with a specific exit value.
- It takes as input the **exit value** you want the process to exit with:
  - **0** for SUCCESS
  - **1** (or any other integer) for ERROR
- If your script cannot continue due to an error, **you should handle the error and/or `exit 1`**.  
Otherwise the code may continue running without the required information.
- If your script has encountered **no errors**, it is nice to `exit 0` so that the user knows everything ended well.  
However, to simplify writing scripts, the default exit code is actually 0 and `exit 0` is actually equivalent to `exit` with no parameters.  
So in most cases **you do not need to write it at all**.

# Taking decisions: if syntax and conditions

If you have never heard of binary logic, it is good to watch or read the *Binary System* material in AdvancedTopics in Lecture3 on Canvas.

- Enable the machine to **decide** on actions depending on certain **conditions**. (**if...then...else...fi**)
- A condition is usually a **command exit value**, it can be specified in many ways but for this part of the tutorial we will consider the command “extended test command” syntax:

```
[[ <expression> ]]
```

- The BASH **if** syntax is as follows:

```
if [[ <expression> ]]; then
    <command1>; [<command2>;...]
else
    <commandA>; [<commandB>;...]
fi
```
- Let's have a closer look at those two.

# Basic conditions

- `[[ <expression> ]]` where `<expression>` is usually something like:  
`[<operand>] <comparison operator> <expression> ...`
- Example expressions:
- just the number 3  
`[[ 3 ]]`
- is 3 equal to 2?  
`[[ 3 == 2 ]]`
- `#` does the file filename exist?  
`[[ -e filename ]]`
- 0, 1, 3, 2, filename are **operands**
- `==`, `-e` are **comparison operators**.



# Basic conditions

- A condition statement is nothing but a command itself. It returns an exit value like any other command.
  - This is quite weird because we have the following:
    - 0 means TRUE
    - 1 means FALSE

Which can be misleading if you remember about boolean expressions in Lecture 3!

- Example:

```
# is 3 equal to 2 ?
[pflorido@pptest-iridium tutorial3]$ [[ 3 == 2 ]]
# we test the exit value with echo:
[pflorido@pptest-iridium tutorial3]$ echo $?

1

# 1 means the test failed. 3 not equal to 2

# Let's test if 3 is equal to 3...
[pflorido@pptest-iridium tutorial3]$ [[ 3 == 3 ]]
[pflorido@pptest-iridium tutorial3]$ echo $?

0

# 0 exit value means: yes, the condition is true, 3 is equal to 3
```

# most used comparison operators

## • Files

- True if filename exists  
`[[ -e <filename> ]]`
- True if a file exists and it is a simple file  
`[[ -f <filename> ]]`
- True if a file exists and it is a directory  
`[[ -d <dirname> ]]`

## • Strings

- True if a string has length 0:  
`[[ -z <string> ]]`
- True if a string is equal to another  
`[[ <string1> == <string2> ]]`
- True if a string is different from another  
`[[ <string1> != <string2> ]]`
- One of the things that are hard to achieve is **to check if a variable has been defined and it is empty**. Without explaining the reasons why this is not easy, I'll just show you the most common trick to test if a variable it's empty:
  - True if the variable VAR is empty.  
`[[ "x$VAR" == "x" ]]`  
What the above does is: it concatenates x to the content of VAR. If var it's empty, then `x == x` it's true.
- A full list can be found at
  - [https://www.gnu.org/software/bash/manual/html\\_node/Bash-Conditional-Expressions.html](https://www.gnu.org/software/bash/manual/html_node/Bash-Conditional-Expressions.html)
  - <https://tldp.org/LDP/abs/html/comparison-ops.html>

# Bash conditions and the binary logic

- The two operators:

- && (logical AND)
- || (logical OR)

Do not work exactly as one would expect.

- `[[ 0 && 1 ]]` will always return `$?=0`, no matter where you put 0 or 1
- To do arithmetics and binary logic it's better to use the "*arithmetic expansion*" command `$(( ))`
  - `(( 0 && 0 ))` will exit 0
  - `(( 0 && 1 ))` will exit 1
  - `(( 1 && 0 ))` will exit 1
  - `(( 1 && 1 ))` will exit 1These are consistent with binary logic, but we will not cover them in this tutorial.

# Bash conditions and the binary logic

- The `&&` and `||` operators are more commonly used to chain commands with the principle of *lazy evaluation*, that is, since we know that FALSE AND (whatever) it is always FALSE, then the `&&` just checks the first operand to decide the result. It only goes to the second if needed. `||` does the same with TRUE.
- Example usage:
  - `command1 && command2`
    - If `command1` fails, then `$? = 1` and `command2` is **not** executed
  - `[[ $FILENAME != 'wrongfilename' ]] && [[ -e $FILENAME ]]`  
(the `$FILENAME` variable does not contain `wrongfilename`)  
AND  
(there is a file whose name is the value contained in `$FILENAME`)
    - if `$FILENAME` contains `wrongfilename` then the first condition is false, and the second condition is not checked due to lazy behavior.  
The whole statement returns `$?=1` (false)  
it is enough that one condition is false for the whole statement to be false.
  - `command1 || command2`
    - If `command1` fails, then `$? = 1` but `command2` is also executed
  - `[[ $FILENAME != 'wrongfilename' ]] || [[ -e $FILENAME ]]`  
(the `$FILENAME` variable does not contain `wrongfilename`)  
OR  
(there is a file whose name is the value contained in `$FILENAME`)
    - if `$FILENAME` contains `wrongfilename` then the first condition is false, second condition is checked, and only if `$FILENAME` does NOT exist then whole statement returns `$?=1` (false)  
both conditions must be false for the whole statement to be false.

# Exercises

- **Exercise 3.13:** Find the definition of the condition statements below on this page:  
<https://tldp.org/LDP/abs/html/comparison-ops.html>
- And test the exit values (`echo $?`) of the following condition statements:
- `[[ -e /etc ]]`
- `[[ -e /doesnotexist ]]`
- `[[ 15 -lt 15 ]]`
- `[[ 16 -lt 15 ]]`
- `[[ 15 -eq 15 ]]`
- `[[ 15 == 15 ]]`
- `[[ 'large' != 'small' ]]`
- `[[ 'large' -ne 'small' ]]`
- Bash has also data types, but they are not explicit. In fact you cannot use `-ne` with strings, only with numbers (integers). This is why the last statement will return 1.

# Control structures:

## if ... then ... else .. fi 1/2

- Consider the example below:  
( -le is the operator “less than or equal”)

```
#!/bin/bash
# testif.sh
# run with: ./testif.sh arg1 arg2 arg3
#
# test that at least three arguments are passed to the script

if [[ $# -le 2 ]]; then
    echo "Not enough arguments. Must be at least 3!";
    # exit with error, not zero
    exit 1;
else
    echo "More than 2 arguments. Good!";
    # exit without error, zero
    exit 0;
fi
```

- The IF statement will execute the code between **then** and **else** if and only if the condition `[[ $# -le 2 ]]` is TRUE ( $\$? = 0$ )
- Otherwise it will execute the code between **else** and **fi**
- It is possible to write an if statement without the else :  

```
if [[ <expression> ]]; then
    <commands>
fi
```

# Control structures:

## if ... then ... else .. fi 2/2

- **Exercise 3.14:** test the above code with 1 or 3 arguments and check that the exit value is consistent with the code.

```
./testif arg1; echo $?
```

```
./testif arg1 arg2 arg3; echo $?
```

- **Exercise 3.15:** Based on the previous example `testif.sh`, add to the script an if that checks if the first two parameters are the same string.

If they are returns an error.

Example output:

```
[pflorido@pptest-iridium solution]$ ./testif.sh arg1 arg2 arg3
More than 2 arguments. Good!
Good: the first two parameters differ.
[pflorido@pptest-iridium solution]$ ./testif.sh arg1 arg1 arg3
More than 2 arguments. Good!
Error: the first two parameters are the same string
```

# BASH power:

## Command substitution and pipe

- Bash can capture the output of a command and you can reuse it in a script with a syntax that is called “command substitution”
- Consider the `captureoutput.sh` script.

```
#!/bin/bash

# capture the output of the id command in a variable
USERINFO=$(id pflorido)

# extract the main group by using pipe and the cut command
MAINGROUP=$(id pflorido | cut -d' ' -f 2)

echo "USERINFO is: $USERINFO"
echo "MAINGROUP is: $MAINGROUP"

# same as above but using the pipe on the variable content
echo "MAINGROUP reusing the \$USERINFO variable content:"
echo "$USERINFO" | cut -d' ' -f 2
```

- Notice the use of the `$( ... )` construct. It is used to *capture* the output of the commands between parentheses.
- Notice the use of the `|` (*pipe*) symbol. It sends the output of the `id` command as the input to the `cut` command
- Likewise, one can `echo` the content of the variable `$USERINFO` and pipe it as the input of `cut`
- More about command substitution:  
[https://www.gnu.org/software/bash/manual/html\\_node/Command-Substitution.html](https://www.gnu.org/software/bash/manual/html_node/Command-Substitution.html)



# escaping variables and the cut command

```
#!/bin/bash
...
# same as above but using the pipe on the variable content
echo "MAINGROUP reusing the \${USERINFO} variable content:"
echo "${USERINFO}" | cut -d' ' -f 2
```

- Note the use of the backslash `\` to *escape* the special character `\$` :  
This prevents echo to interpret `$USERINFO` as a variable value
- `cut` is a command that divides a string in fields based on a field separator (`-d' '`, a space) and takes the n-th field (the 2<sup>nd</sup> in this example). More about cut at:
  - <https://www.geeksforgeeks.org/cut-command-linux-examples/>
- **Exercise 3.16:** Run the script `captureoutput.sh` and check the output against these slides to understand what it does. Modify the script for `cut` to print just the user numerical ID.

# Bash Tutorial part 2

- Functions
- Environment, Binding and Scope
- Customizing your environment
- Conditions
- Control structures: loops
- Useful commands

# Functions

- Sometimes we need to do the same task a certain number of times, and it's a bit boring to copy paste. Consider the following example `getsyslines.sh` where we want different lines from different files to get an idea of the system we're running: cpu. memory, network name (hostname):

```
#!/bin/bash

# put the first two lines of /proc/cpuinfo in CPUINFO
CPUINFO=$(cat /proc/cpuinfo | head -2)
# write the content of CPUINFO to screen
echo "First 2 lines of /proc/cpuinfo:"
echo "$CPUINFO"

# put the first four lines of /proc/meminfo in MEMINFO
MEMINFO=$(cat /proc/meminfo | head -4)
# write the content of MEMINFO to screen
echo "First 4 lines of /proc/meminfo:"
echo "$MEMINFO"

# put the first line of /etc/sysconfig/network in HOST
HOST=$(cat /etc/sysconfig/network | head -2)
# write the content of MEMINFO to screen
echo "First 2 lines of /etc/sysconfig/network:"
echo "$HOST"
```

# Functions – identifying parameters

- When you have code like this, it's good to identify similarities that could be parameters to a function: can we simplify the code?

```
#!/bin/bash

# put the first two lines of /proc/cpuinfo in CPUINFO
CPUINFO=$(cat /proc/cpuinfo | head -2)
# write the content of CPUINFO to screen
echo "First 2 lines of /proc/cpuinfo:"
echo "$CPUINFO"

# put the first four lines of /proc/meminfo in MEMINFO
MEMINFO=$(cat /proc/meminfo | head -4)
# write the content of MEMINFO to screen
echo "First 4 lines of /proc/meminfo:"
echo "$MEMINFO"

# put the first line of /etc/sysconfig/network in HOST
HOST=$(cat /etc/sysconfig/network | head -2)
# write the content of MEMINFO to screen
echo "First 2 lines of /etc/sysconfig/network:"
echo "$HOST"
```

# Functions - definitions

- One can define functions to reduce complexity and increase readability

```
# function definition
myfunction() { echo "this is the body of the function" }
# function call
myfunction
```

- A bash function has:

- A **name**, so that is possible to reuse the function, usually followed by two parentheses **()**;
  - Example: `myfunction()`
- A **definition**, where the operations that the functions will do are defined. It is also called the **body** of the function.
  - The body of the function **MUST** be enclosed in curly brackets **{ }**. These delimit a **block of code**
  - The body of the function is executed **ONLY** when the function is *called*, **not** when it is defined.
  - Example: `{ echo "this is the body of the function" }`
- **Parameters**, that are handled the same as command line arguments with the predefined variables `$#`, `$0`, ..., `$n`. `$0` is the name of the function!  
Example: `{ echo "the first parameter is $1" }`
- Several **calls**. A call is when the name of the function appears with parameters to the function.
  - The call will trigger an **instantiation** of the parameters inside the body of the function, that is, the values of the `$1`, `$2` variables will be *substituted with the parameters*.
  - the function **body will be executed** with the values of the *passed* parameters.  
Example: `myfunction param1`  
`myfunction param2 param3`  
`myfunction param...`

# Functions – example refactored

- Please take your time to look at the *refactored* code for `getsyslines.sh`, `getsyslines_function.sh`:

```
#!/bin/bash

# Function DEFINITION:
# Function that takes in input a filename and a number of lines
# outputs a message about the printed lines
# function NAME
printlinesoffile()
{ # start function BODY
  # the first parameter is a filename
  FILENAME=$1
  # the second parameter is a number of lines
  NUMLINES=$2

  # RESULT is a variable with side effect: the result is stored
  # in a global variable
  # be CAREFUL when to extract the value outside the function!
  # It changes at every function call!
  RESULT=$(cat $FILENAME | head -$NUMLINES)

  # Print out the lines
  echo "First $NUMLINES line(s) of $FILENAME:"
  echo "$RESULT"
} # end of function BODY

# function CALL: put the first two lines of /proc/cpuinfo in CPUINFO
printlinesoffile /proc/cpuinfo 2
CPUINFO=$RESULT

# function CALL: put the first four lines of /proc/meminfo in MEMINFO
printlinesoffile /proc/meminfo 4
MEMINFO=$RESULT

# function CALL: put the first two lines of /etc/sysconfig/network in HOST
printlinesoffile /etc/sysconfig/network 2
HOST=$RESULT
```

# Side effects

- Mathematical functions only return values.
- A programming language function or procedure not only returns a value, but usually changes the **environment** of the process running. This is usually called a **side effect**.
- The content of `$RESULT` is a *side effect* of the `printlinesoffile()` function as it changes the *environment* of the process at every function *call*
- In bash one could say that the exit value `$?` is the return value of the process, and all the produced output is the side effect: it modifies the screen, the filesystem, the memory.... side effects in programming languages are the main reason for programming to be useful.

# Environment, binding

See also the clip about *Bash Environment* on Canvas.

- **Environment:** All the variable and function names “live” in a space called **environment**. You can think of it **as a table** in the compiler or interpreter memory containing all variable names and their associations with memory chunks.
- **Binding:** A name is said to be **bound** to that environment when its value is associated to a memory index in that environment. In the table below we can see some bindings.
  - Binding can be:
    - **Static**, that is, decided at **compile time**
    - **Dynamic**, that is, decided at **runtime**  
(yes one can change where in the memory that variable is pointing)
  - When we **define** a variable or a function, the variable/function name is **added** to the **environment**

Environment	Variable name	Value
global	PWD	Current dir
global	SHELL	Current shell
global	PATH	Executable paths
<code>cpuinfo.sh</code>	CPUINFO	First 2 lines of /proc/cpuinfo
<code>getsomelines_function.sh</code>	RESULT	18458
<code>getsomelines_function.sh</code>	printlinesoffile()	3515



# Visibility, scope

- A variable is **visible** in an environment when its binding is present in that environment, that is:
  - There **exists** a variable **name** in the environment
  - That variable name is **associated to a memory location** (this depends on languages)
- Usually a function has its own environment, that is, a set of variables in its own environment, and can see the variables in other environments according to some rules. These rules define the **scope**, or **visibility**, of a variable.
- In the case of BASH, **functions do not have own environment.**

The scope or visibility of a variable in bash is **limited to a bash instance and all its children**. Let's see some examples.

- In BASH there are two kinds of environment:
  - The **set** environment, which only belongs to a running process;
  - The **export** environment, which is a subset of the set environment which is *exposed* to child processes, or processes run inside the same bash.

# Exercise: inspect the environment

## • Exercise 3.17:

- `set` and `export` are two bash builtin commands to inspect the environment.
  - print out the set environment with the `set` command. This contains both exported, inherited and temporary variables.
  - print out the export environment with the `export` command. This contains only exported variables, no functions.
- `env` is a stand alone command that works with all the shells.
  - print the environment using the `env` command. This contains the default environment an application is ran with when executed outside bash.

## • Exercise 3.18 :

- find some functions from the environment: we know that a function name is followed by (). Hence we can do:
  - `env | grep '()'`
  - `set | grep '()'`
- `grep` is a command that finds a string in a file.

# The BASH environment: export

Everytime one opens a terminal, the program bash is executed and a **new environment** is created.

1. Open a terminal.

2a. Run the **set** command. You'll see all the variables in the current bash session.

Everytime a variable is initialized it ends up in the set environment.

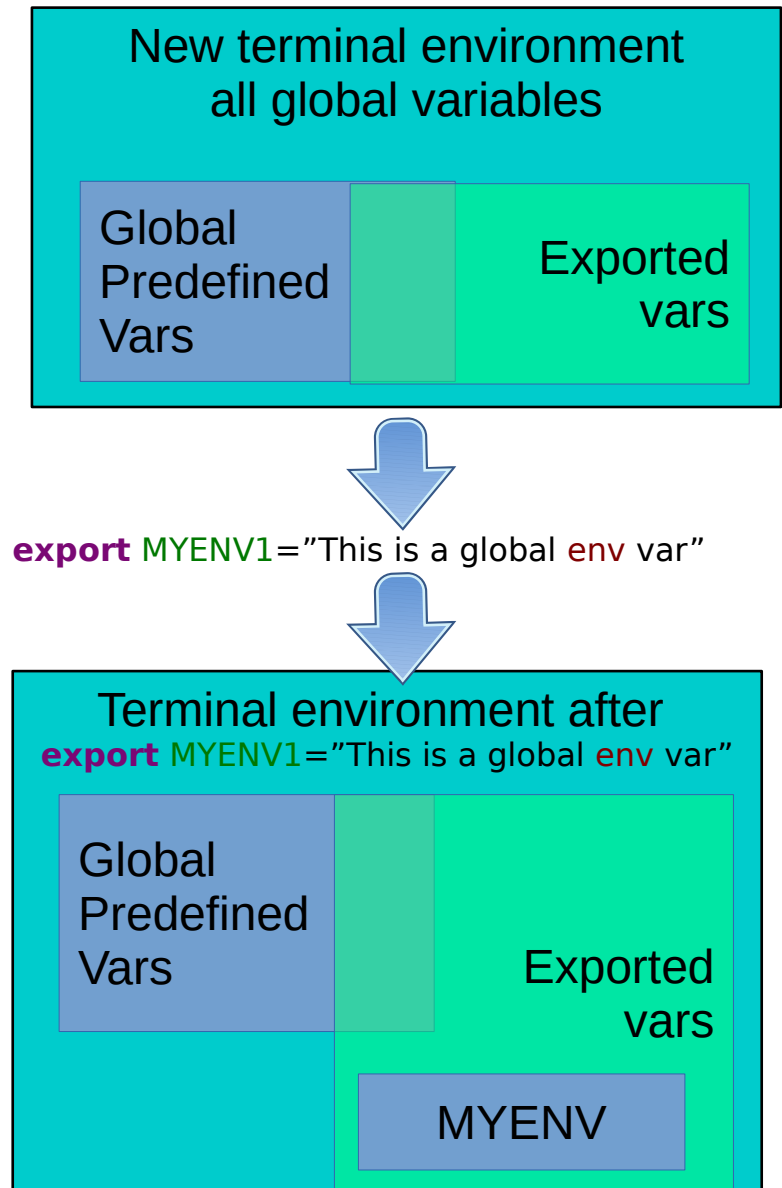
2b. Run the **export** command. You'll see all the environment variables in the current bash session that will be **exported** to any child process.

3. Create and initialize a new **exported** environment variable:

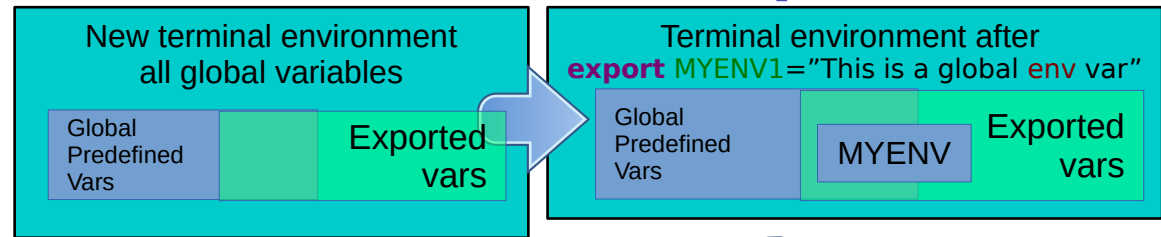
```
export MYENV1="This is a global env var"
```

4. Search for the variable after running **export**, or just print its content with

```
echo $MYENV1
```



# The BASH environment: export



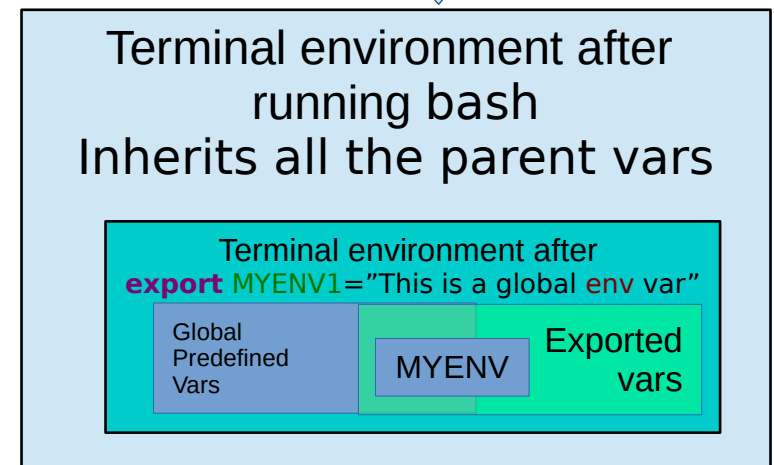
5. Now open another bash instance:

- Write the command **bash** and press enter. You are now in a new bash command line.
- Run the command **export**. You will find that `MYENV1` is still there.  
The environment is said to be **inherited** from the father process.
- **This happens every time you start a bash script =>** Starting a bash script is equivalent to executing the command bash and then a sequence of commands.

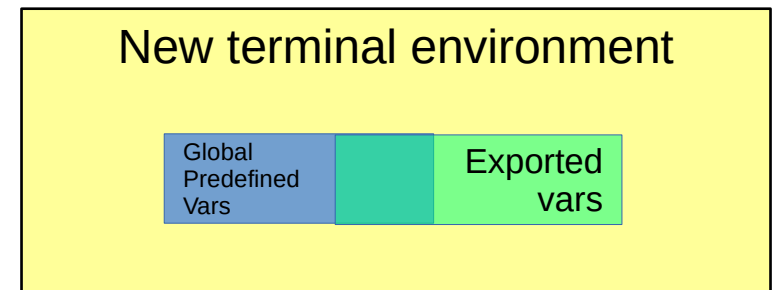
6. Open another terminal and run **export echo \$MYENV1**

- `MYENV1` should not be there.  
**There is no environment inheritance between terminal windows.**
- Switch back to the terminal where `MYENV1` is defined.

Execute "bash"



≠



# BASH environment: scope 1/3

- **Exercise 3.19 (see clip on Canvas):** Consider the bash script `envtest.sh` in the examples/ **folder** with the following content:

```
#!/bin/bash

# test if an environment variable is defined
if [ "x$MYENV1" == "x" ]; then
echo "MYENV1 not defined in the environment or empty. Please run"
    echo '  export MYENV1="This is my first environment variable. This is exported to all children
processes"'
    echo
fi

# create an environment variable. Adds only to the environment of this script
MYENV2="This is my second environment variable, MYENV2 is defined only in this process"

# export and environment variable. Adds to the environment where this script is ran
export MYENV3="This is my third environment variable, exported. MYENV3 is defined in this process and in all
children processes"

# write the content of the environment vars to screen
echo "Content of MYENV1: $MYENV1"
echo "Content of MYENV2: $MYENV2"
echo "Content of MYENV3: $MYENV3"

echo "Now check if MYENV2 and MYENV3 contents are still defined, with the commands:"
echo '  echo "Content of MYENV1: $MYENV1"'
echo '  echo "Content of MYENV2: $MYENV2"'
echo '  echo "Content of MYENV3: $MYENV3"'
echo
```

# BASH environment: scope 2/3

- I export MYENV1 as in the example before, run ./envtest.sh and check which variables are defined:  
MYENV2 and MYENV3 exist only inside the script, they're in the scope of the script but not outside. They disappear once the script finishes.

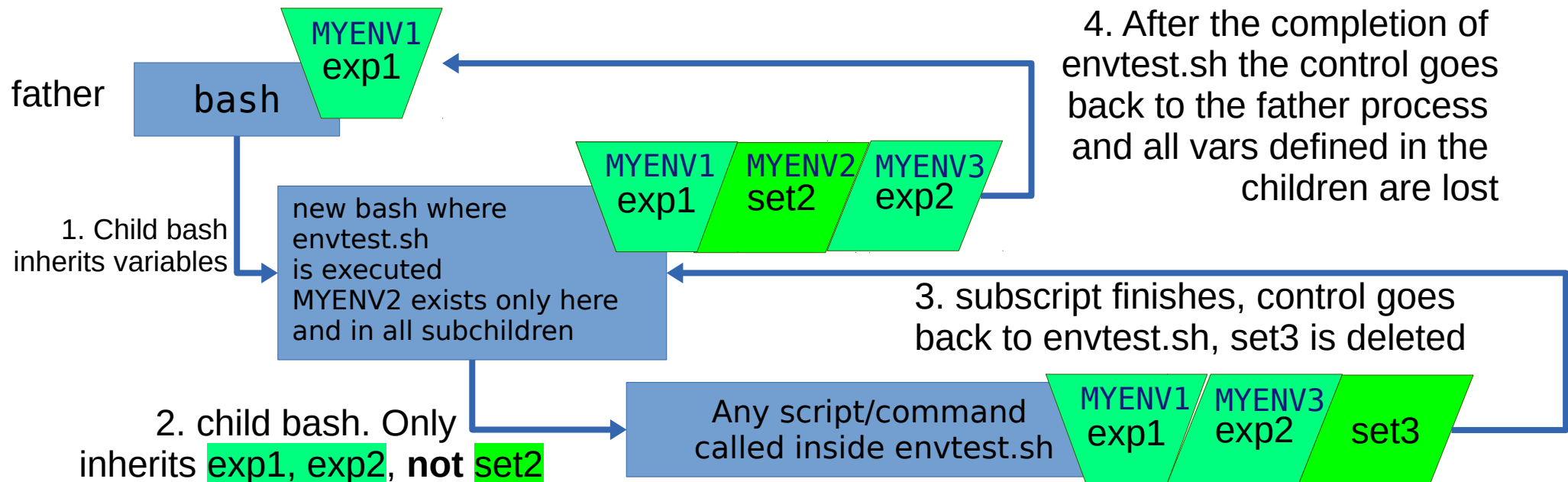
```
[pflorido@pptest-iridium ex3.19]$ export MYENV1="This is my first environment
variable. This is exported to all children processes"
[pflorido@pptest-iridium ex3.19]$ ./envtest.sh
Content of MYENV1: This is my first environment variable. This is exported to all
children processes
Content of MYENV2: This is my second environment variable, MYENV2 is defined only in
this process
Content of MYENV3: This is my third environment variable, exported. MYENV3 is defined
in this process and in all children processes
Now check if MYENV2 and MYENV3 contents are still defined, with the commands:
    echo "Content of MYENV1: $MYENV1"
    echo "Content of MYENV2: $MYENV2"
    echo "Content of MYENV3: $MYENV3"

[pflorido@pptest-iridium ex3.19]$ echo "Content of MYENV1: $MYENV1"
Content of MYENV1: This is my first environment variable. This is exported to all
children processes
[pflorido@pptest-iridium ex3.19]$ echo "Content of MYENV2: $MYENV2"
Content of MYENV2:
[pflorido@pptest-iridium ex3.19]$ echo "Content of MYENV3: $MYENV3"
Content of MYENV3:
```

# BASH environment: scope 3/3

- When you run a script, a new bash instance is generated for the script, that inherits the father environment
- Once the script finishes, all variables defined or exported **inside the script** are **cleared from the environment table** and the control goes back to the father process.

The “father” environment (where you ran the bash command) DOES NOT inherit from “children” (executed script), but bash scripts executed inside it have their own environment that **inherits** from the father.



# Importing an environment changes the scope 1/4

- In bash, there is a command that allows you to copy the environment defined in a script to another script or bash instance, so that it survives the termination of a script. This command is **source**
- **Careful! The command also executes EVERYTHING inside the BASH script!**
- If you now try
  - **source ./envtest.sh**
  - You'll see that the output of **export** will contain: **MYENV1, MYENV2 and MYENV3**
    - **MYENV2** is only in the **set** or local environment, but sourced from the script
    - **MYENV1, MYENV3** are in the **export** environment and they will be copied to every child process.

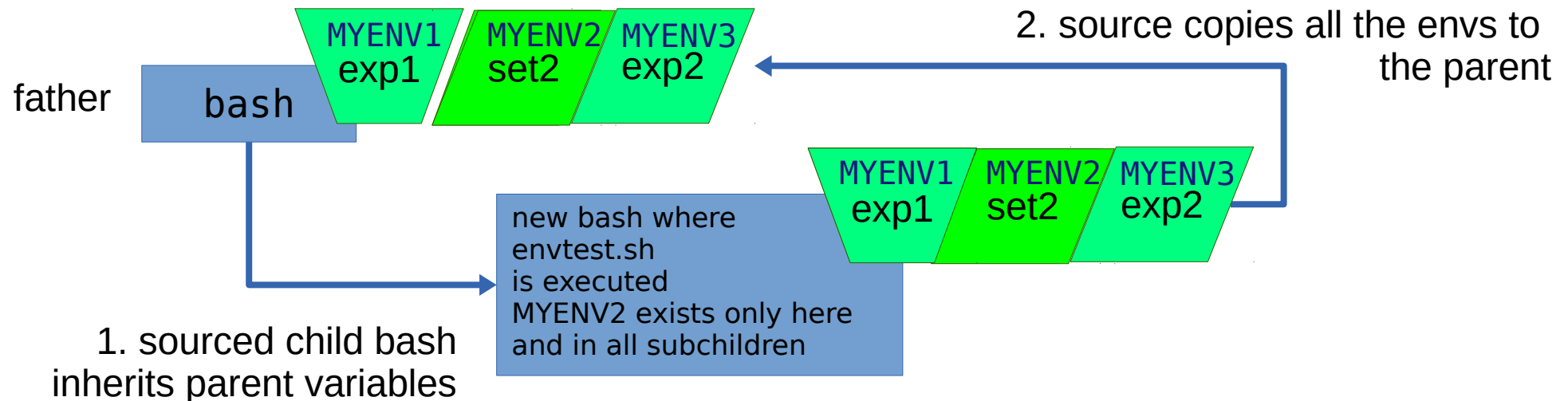
```
[pflorido@pptest-iridium ex3.19]$ source envtest.sh
Content of MYENV1: This is my first environment variable. This is exported to all children processes
Content of MYENV2: This is my second environment variable, MYENV2 is defined only in this process
Content of MYENV3: This is my third environment variable, exported. MYENV3 is defined in this
process and in all children processes
Now check if MYENV2 and MYENV3 contents are still defined, with the commands:
echo "Content of MYENV1: $MYENV1"
echo "Content of MYENV2: $MYENV2"
echo "Content of MYENV3: $MYENV3"

[pflorido@pptest-iridium ex3.19]$ echo "Content of MYENV1: $MYENV1"
Content of MYENV1: This is my first environment variable. This is exported to all children processes
[pflorido@pptest-iridium ex3.19]$ echo "Content of MYENV2: $MYENV2"
Content of MYENV2: This is my second environment variable, MYENV2 is defined only in this process
[pflorido@pptest-iridium ex3.19]$ echo "Content of MYENV3: $MYENV3"
Content of MYENV3: This is my third environment variable, exported. MYENV3 is defined in this
process and in all children processes
```



# Importing an environment changes the scope 2/4

- When *sourcing* a (child) script, all the variables declared in the script are copied in the current (parent) environment.



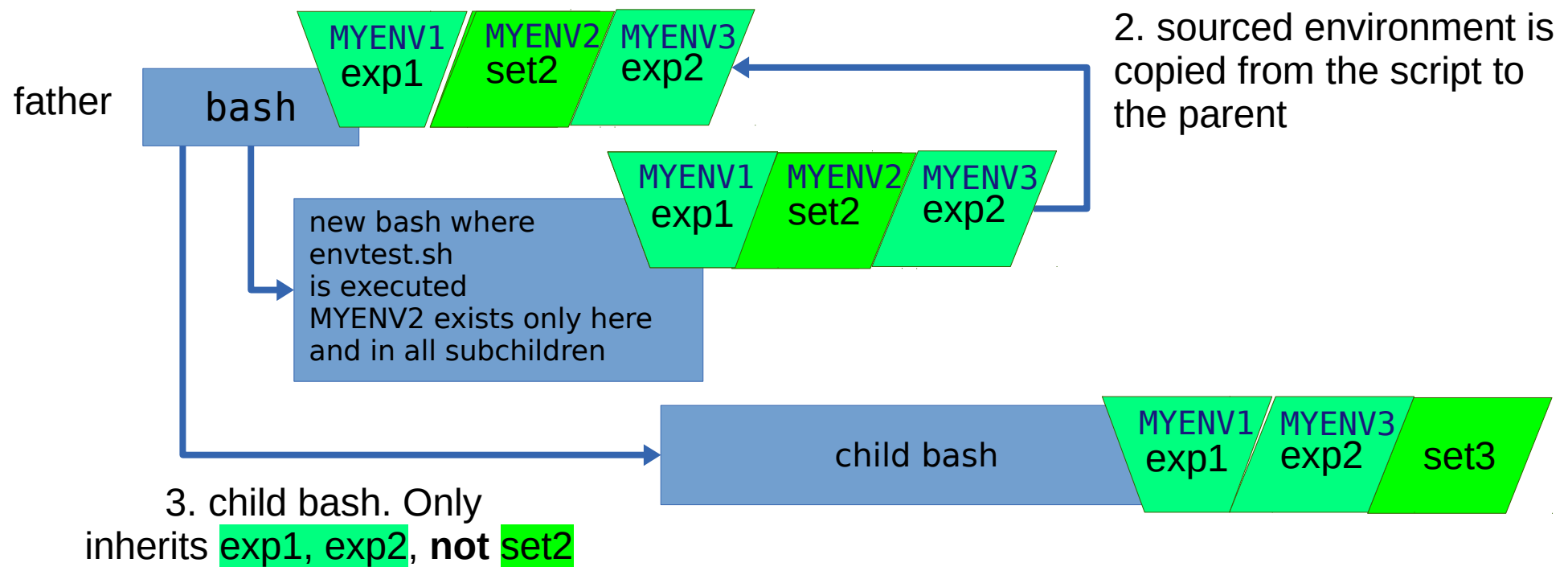
# Importing an environment changes the scope 3/4

- If you now write **bash** and open a **child bash process**, you will see that MYENV2 is empty. It is actually not defined anymore, it is lost inside the parent process because it was part of the **set environment**.
- Only the **exported environment** survives – which is why in bash the exported environment is usually the one referred as **the environment**.

```
[pflorido@pptest-iridium ex3.19]$ bash
bash-4.1$ echo "Content of MYENV1: $MYENV1"
Content of MYENV1: This is my first environment variable. This is exported to
all children processes
bash-4.1$ echo "Content of MYENV2: $MYENV2"
Content of MYENV2:
bash-4.1$ echo "Content of MYENV3: $MYENV3"
Content of MYENV3: This is my third environment variable, exported. MYENV3 is
defined in this process and in all children processes
```

# Importing an environment changes the scope 4/4

- The child of a sourced environment inherits only the **exported environment** of the father process



# Clearing the environment

- One can clear the variables added to the environment using the **unset** command.  
Example:

```
[pflorido@pptest-iridium ex3.19]$ echo "Content of MYENV1: $MYENV1"
Content of MYENV1: This is my first environment variable. This is exported to all children
processes
[pflorido@pptest-iridium ex3.19]$ echo "Content of MYENV2: $MYENV2"
Content of MYENV2: This is my second environment variable, MYENV2 is defined only in this
process
[pflorido@pptest-iridium ex3.19]$ echo "Content of MYENV3: $MYENV3"
Content of MYENV3: This is my third environment variable, exported. MYENV3 is defined in
this process and in all children processes
[pflorido@pptest-iridium ex3.19]$ unset MYENV1 MYENV2 MYENV3
[pflorido@pptest-iridium ex3.19]$ echo "Content of MYENV1: $MYENV1"
Content of MYENV1:
[pflorido@pptest-iridium ex3.19]$ echo "Content of MYENV2: $MYENV2"
Content of MYENV2:
[pflorido@pptest-iridium ex3.19]$ echo "Content of MYENV3: $MYENV3"
Content of MYENV3:
```


# Environment summary

- Every **new terminal** window creates a **new environment**. Environments are **not** shared within terminal windows.
- An initialized variable only “exists” in the environment of the bash instance where it was *initialized*.
- To make sure a variable survives in all script launched inside a bash instance, one must **export** it
  - Exported variables are only inherited by child processes and not by parent processes.
- One can **import** the environment that a script generates by using the **source** command
  - Remember: do **not** write any **exit** in code you plan to *source*!
- The variables in the export environment are commonly called **environment variables**.
- The environment variables can be cleared using the **unset** command.

# Customizing your environment

- When opening a terminal or starting bash, there are a few key files that are processed to *initialize* your shell environment.
- Depending on the distribution and the shell, these may vary. Some are *system* files and you cannot change them, these are processed **first** when opening a shell. But you can override them inside your *user files*, that are processed after the system ones.
- System files:
  - `/etc/profile`
  - All files in `/etc/profile.d/`
  - `/etc/bash.bashrc`
- User files. These are hidden, hence their names starts with a dot. You can see them with `ls -a ~`
  - `~/.profile`
  - `~/.bashrc`
  - `~/.bash_profile`
- You can inspect the content of those files using `cat`, `less` or `gedit`. Ask me about things you do not understand.
- IMPORTANT: `.bashrc` should **NEVER** contain code that generates **output** when `.bashrc` is executed.

# Customizing your default environment - exercise

- We will add
  - A variable containing Aurora's folder for the course, MNXB01DIR
  - an *alias* - kind of a macro or shortcut - to the `cd` command, `cdmnb01`, that allows us to quickly access the shared folder.
- The `alias` command is used for that. Try it and you will see the list of active aliases.
- **Exercise 3.20 - add cdmnb01 alias**
  - 1. backup your existing `.bashrc` file:
    - `cp ~/.bash_profile ~/.bash_profile20200911backup`
  - 2. Open `.bash_profile` with *Pluma*
    - `pluma ~/.bash_profile &`
  - 3. Add at the end of the file the following lines:
    - `MNB01DIR=/projects/hep/fs10/mnb01`
    - `alias cdmnb01='cd $MNB01DIR'`
  - 4. Import the newly created alias by sourcing the new bashrc:
    - `source ~/.bash_profile`
  - 5. It should now appear in the list if you write `alias`
  - 6. Test that you can use the newly added `cdmnb01` command! It will move you directly to the shared folder in Iridium.
  - You can even see it when you press tab 

# More on control structures: loops

- Allow the code to **loop until a certain condition** is met (**while . . . do . . . done**)
- Allow the code to **loop** for a definite number of times or **over a list** of objects (**for . . . do . . . done**)



# Control structures:

## for ... do ... done

- Repeat something for a predefined number of times or for each element in a list.
- Syntax:  
**for** <i> **in** <list>; **do**  
    <command1>; [<command2>;...]  
**done**
- The interpreter will substitute <i> with an element in <list> inside the code block **do** ... **done** and execute the code for each element.

# Control structures: for ... do ... done

- Lists passed to `for` can be defined in many ways:
- plain list: strings separated by a space  
Example: `a b c d`
- intervals: `{1..10}` all numbers from 1 to 10.  
It expands to the list  
`1 2 3 4 5 6 7 8 9 10`

```
#!/bin/bash
# forexamples.sh
# run with: ./forexamples.sh
#

echo "print three strings:"
for word in one two three; do
    echo "$word"
done

echo "Countdown starting:"
for i in {1..10}; do
    echo "now counting $i"
    # wait 1 second
    sleep 1
done

# ok this is just for fun
banner boom!
```

# Control structures: for ... do ... done

- The wildcards like \* result in a list of files separated by spaces, hence can be used to do operations on a directory of files.
- The following prints types of files in some directory, defaults to the home directory ~

```
#!/bin/bash
# listfiletypes.sh
# run with: ./listfiletypes.sh <directory>
#
# Print the argument values
TARGETDIR=$1

# A typical use of IF: if no TARGETDIR defined, then # x == x and the expression in brackets will be false, so the else branch
# will be executed and an error message will be shown.
if [ "x$TARGETDIR" == "x" ]; then
    TARGETDIR=~
    MESSAGE="No argument found. Listing filetypes for the $TARGETDIR directory by default"
else
    MESSAGE="Scanning filetypes for the ${TARGETDIR} directory"
fi

echo "$MESSAGE"

# scan all files in TARGETDIR
for somefile in ${TARGETDIR}/*; do
    echo "This is the file $somefile, with type:";
    # the file command tells you the type of a file.
    file $somefile
done
```

# Control structures:

## `while ... do ... done`

- Keeps doing something as long as *<condition>* is satisfied.
- Syntax:  
**while** *<condition>*; **do**  
    *<command1>; [<command2>; ...]*  
**done**
- The code contained inside **do ... done** keeps being executed. It will stop when *<condition>* is false.

# Control structures: while ... do ... done

- Ask the user to enter a variable value (using the read command) until the string end is entered

```
#!/bin/bash
# testwhile.sh
# run with: ./testwhile.sh
#
# Continue asking numbers until the user writes "end"

while [ "$var1" != "end" ]; do      # while test "$var1" != "end"
    echo "Input variable value (end to exit) "
    read var1                      # Not 'read $var1' (why?).
    echo "variable value = $var1"  # Need quotes because of "#" . . .
    # If input is 'end', echoes it here.
    # Does not test for termination condition until top of loop.
done
exit 0
```

# Control Structures: Exercises

- **Exercise 3.21:** Run `forexamples.sh` and understand what it does. Reuse the code in `forexamples.sh` code to write a bash script that counts 2 4 8 16 BOOM! (you can keep `sleep 1`, doesn't matter!)
- **Exercise 3.22:** Run `testwhile.sh` and understand what it does. Change the `testwhile.sh` code to bannerize (using `banner`) any word the user inputs and exits when the user writes STOP!
- **Exercise 3.23:** Improve the above with that the code exits without printing STOP! as a banner, but prints a message when STOP! has been typed by the user. Using a function and variable might help.

# Bash “Libraries”

- Now that we know what the environment is and how to source a script, we have all the ingredients to create *bash libraries*
- A *bash library* is just a bash script with the following features:
  - No code is printed out when executing the script
  - It may contain only initialization of variables and definitions of functions
    - But be careful that variables will be added to the existing environment and will overwrite all variables with same names!
  - For other scripts to use it, these script must **source** it
  - The bash library script must be in a location known to all the scripts that source it.
  - It doesn't need to start with `#!/bin/bash` since the calling scripts are supposed to be bash scripts.
  - It doesn't contain any **exit** statements if possible
- It is best if the library is sourced as early as possible in the code to avoid variable overwrite.

# Bash library example

- An example of libraries is in the homework for 2021. I did not have time to develop good examples/exercises.

<https://github.com/floridop/MNXB01-2021/blob/main/floridopag/tutorial3/homework3/solution/cityandyear.sh>



# Additional material for reference

# More on conditions

- Conditions are of different kinds depending on the languages.  
**The only condition that BASH can check is whether a command execution terminates successfully.**
  - An exit value of **0** is **TRUE (termination successful)**, all **other values** are **FALSE (termination unsuccessful)**.
- The way to specify conditions is as follow:
  - The square bracket **[ ]** or the **test** command can be used.  
Documentation: **man test**
    - Example: **test -e <filename>** checks if a file exists; if the file exists, the predefined variable **\$?** will contain 0, otherwise 1.
    - Try **echo \$?** after running a test to see the exit value of the test command.
  - The double square bracket or extended test **[[ <some test command> ]]**  
Documentation: execute **man bash** and search for “conditional expression”
    - Example: **[[ -e /etc/services ]]**
  - The double parentheses for arithmetical expansion and logical operations.  
**<a>** and **<b>** should be integers.  
**(( <a> && <b> ))**  
Documentation: execute **man bash** and search for “Arithmetic Expansion”
- Tips:
  - to search while in **man**, type the **/** character followed by a search string and then press Enter.
  - To exit **man**, use the key **q**
  - To move around use the arrows.

# More on conditions: Exercises

- Execute the following commands:
  - The /etc file exists, so test should exit with no errors  
`test -e /etc`
  - Hence the following should be 0  
`echo $?`
  - This file for sure does not exist! It should put an error in the exit status  
`test -e /thisfiledoesnotexist`
  - What is the exit status now? Should be 1, means error, the file did not exist  
`echo $?`
  - The brackets are equivalent to the above. Try!  
`[ -e /etc ]`  
`echo $?`  
`[ -e /thisfiledoesnotexist ]`  
`echo $?`
  - The double brackets are also equivalent for this case, but they can do also logic and arithmetic evaluation if required, which the others above don't.  
`[[ -e /etc ]]`  
`echo $?`  
`[[ -e /doesnotexist ]]`  
`echo $?`

# More on conditions: Exercises

- Execute the following commands. Do you understand the meaning and results? If not, ask me.
  - `true`
  - `echo $?`
  - `false`
  - `echo $?`
- Parentheses are Arithmetic Expansion, and the logical operator `&&` is the boolean AND. Check the lecture on binary system. An important thing to note is the following:
  - **Inside** the parentheses **0=false** and **1=true** which is the **opposite** of bash/linux exit code logic.
  - In bash 0=true 1=false.
  - So the `$?` exit value of boolean false `(( 0 ))` is bash **1**
  - So the `$?` exit value of boolean true `(( 1 ))` is bash **0**
- `(( 0 ))`
- `echo $?`
- `(( 1 ))`
- `echo $?`
- `(( 0 && 0 ))`
- `echo $?`
- `(( 1 && 0 ))`
- `echo $?`
- `(( 1 && 1 ))`
- `echo $?`

# Control structures:

## Alternative for ... do ... done

- Print the arguments using different condition approaches

```
#!/bin/bash
# testfor.sh
# run with: ./testfor.sh arg1 arg2 arg3 ...
#
# Print the argument values

echo "Using lists of elements"
index=1          # Reset argument counter
for arg in "$@"; do
    echo "Arg #$index = $arg"
    let "index+=1"
done              # $@ sees arguments as separate words.

echo "Using C syntax for the condition"
for ((i=1 ; i <= $# ; i++ )); do
    echo "Argument $i is ${!i}";
done
```

- `#$var` forces the content of `var` to be a number
- Parameter substitution  
`${!var}` Gets the **value** of a variable with the name `$var` instead of `var`

# A bit about arithmetic expansion

- The arithmetic expansion context can be used for simple calculations or conditions
- Evaluation (execution of the operation) is done via the `expr` command, but you can avoid it when putting the result in a variable
- on the command line:
  - `expr $(( 5 + 6 ))`  
11
- In a script, you can assign the value to a variable:
  - `VAR=$(( 5 + 6 ))`  
`echo $VAR`  
11

# Case construct

- I add this just for reference, but I do not recommend to use it. It has a lot of unexpected behaviors. It is used when you have a lot of if ... then ... else and you want to avoid writing a lot of them.
- This tutorial is good:  
<https://phoenixnap.com/kb/bash-case-statement>

# References

- Bash scripting:  
<http://tldp.org/LDP/abs/html/>
- Interactive aid:  
<https://explainshell.com>
- GNU bash manual:  
<https://www.gnu.org/software/bash/manual/bash.html>
- A nice collection of things that can go wrong with bash:  
<https://mywiki.woledge.org/BashPitfalls>