Other languages and C++
Writing bash scripts

Florido Paganelli
Lund University
florido.paganelli@hep.lu.se

Outline

- Little theory about C++ environment, binding, scope

- Introduction to scripting
- Bash
  - Scripts
  - Variables in bash: environment, binding, scope
  - Control structures
- Datasets
- Automation using scripting
  - Genesis of an algorithm

Variables, types in C++

- A **variable** is an identifier, a name, for a memory location.
- To **define** a variable is to give a **name** and a **type** to it. This tells the compiler to find a free memory space for that variable.

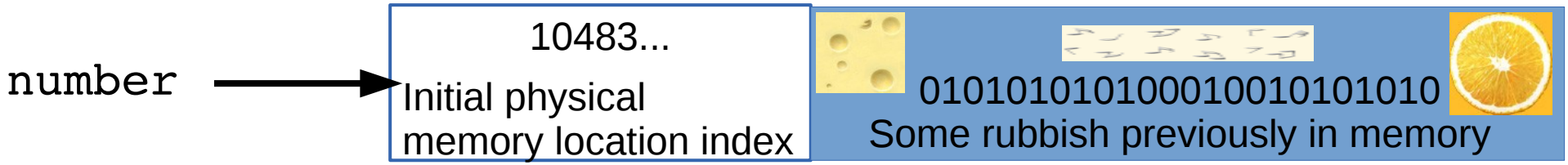
```
int number;
```

- The **type** indicates the kind of information stored inside the variable. In languages like C++ it must be declared explicitly; such languages are also called **typed languages**.
  - The type also defines **the size of the allocated memory**.
  - As the compiler reads your code (*compilation time*), it internally creates table of names of variables with their types, size, tentative memory pointers (**static allocation**).

Var name	Var type	Associated size	Initial tentative logical memory location pointer
`larger`	`int`	`sizeof(int)` `e.g. 2bytes`	10483392805

Variables, types in C++

- If the variable is not **initialized**, it can contain anything. It means that at *runtime*, when the pointer actually will point to a real memory location, whatever is already there will represent the variable **value**.
  - If we were to run the code immediately **without initializing the variable**, we're not sure of what the content of the memory is:



- By **assigning** a **value** to a variable, we tell the compiler what to write in the memory.

```
number = 42;
```

Var name	Var type	Associated size	Initial tentative logical memory location pointer	value
`larger`	`int`	`sizeof(int)` e.g. 2bytes	10483392805	**42**

Environment, binding

- All the variable and function names “live” in a space called **environment**. You can think of it **as a table** in the compiler containing all variable names and their associations with memory chunks.
- 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 on the left we can see some bindings.
- When we **define** a variable, the variable name is added to the **environment**
- In languages like C++ we can see them in the form of **pointers**.
- 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)

Environment	Variable or function name	Starting virtual memory index assigned by compiler (at compile time)	Starting virtual memory index assigned by operating system (runtime)
std	cout	Virt(#200), defined in std	physical(#ABBC)
global			
`global`	foo()	Virt(#1), defined in global	physical(#ABCC)
`foo()`	fooScope	Virt(#2), defined in foo->virt(#1)	physical(#7945)
`foo()`	Anonymous block#1	Virt(#3), defined in foo->virt(#1)	physical(#ABCC)
Anonymous block#1	blockScope	Virt(#4), defined in Anonymous block #1->virt(#3)	physical(#ABCC)

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 C++, **blocks of code** (the curly brackets {}) are used to define new environments and scopes.
  - A variable **defined** in a block is always added to that block environment and **visible** in that block's environment. For ease of use, we say is visible in that block.
    - **Q: What happens if one uses the same names in two blocks???**
    - **A:** The memory to which that name is pointing is overridden by the last block that could change the environment.
      If you don't understand environments and scopes, you will only be able to verify this at runtime.

Functions and scopes in C++

- In C++, the environment and scopes are managed by the use of **blocks of code**.
- The general inheritance rules are as follows:
  - A block **inherits the environment from its parent block**, that is, all the variable and function names existing at the moment of opening the block are **imported** in the block environment.
  - Every variable name **defined** in a block is **added** in the environment of that block.
  - If a variable with the same name is present in the environment, the last defined variable **overrides** any other variable with the same name within that block.
    - That is, it is **not possible anymore to use the value contained in variables with the same name defined outside that block**.

Functions and scopes in C++

```
#include <iostream>
using namespace std;
int globalScope = 0; //This is a global variable, visible everywhere.

void foo() {
   int fooScope = 1; //Only visible within foo function
   cout << "fooScope: " << fooScope << endl;
   cout << "localScope: " << localScope << endl;
}
int main() {
   cout << "globalScope: " << globalScope << endl;

   { //Any block declares a scope, even this useless one
      int localScope = 3;
      cout << "localScope: " << localScope << endl;
      foo();
      cout << "fooScope: " << fooScope << endl;
      int globalScope = 100; // variable hiding, very bad practice!
      cout << "globalScope: " << globalScope << endl;
   }
   cout << "localScope: " << localScope << endl;
   cout << "globalScope: " << globalScope << endl;
}
```

Functions and scopes in C++

```
#include <iostream>
using namespace std;
int globalScope = 0; //This is a global variable, visible everywhere.

void foo() {
   int fooScope = 1; //Only visible within foo function
   cout << "fooScope: " << fooScope << endl;
   cout << "localScope: " << localScope << endl;
}
int main() {
   cout << "globalScope: " << globalScope << end

    { //Any block declares a scope, even this useless
      int localScope = 3;
      cout << "localScope: " << localScope << endl
      foo();
      cout << "fooScope: " << fooScope << endl;
      int globalScope = 100; // variable hiding, very
      cout << "globalScope: " << globalScope << e
    }
   cout << "localScope: " << localScope << endl;
   cout << "globalScope: " << globalScope << end
}
```

Variables in the **global scope** and visible to everyone

Environment	Variable or function name	Parent environment
global	globalScope	

Functions and scopes in C++

```cpp
#include <iostream>
using namespace std;
int globalScope = 0; //This is a global variable, visible everywhere.

void foo() {
   int fooScope = 1; //Only visible within foo function
   cout << "fooScope: " << fooScope << endl;
   cout << "localScope: " << localScope << endl;
}
int main() {
   cout << "globalScope: " << globalScope << end

    { //Any block declares a scope, even this useless
      int localScope = 3;
      cout << "localScope: " << localScope << endl
      foo();
      cout << "fooScope: " << fooScope << endl;
      int globalScope = 100; // variable hiding, very
      cout << "globalScope: " << globalScope << e
    }
   cout << "localScope: " << localScope << endl;
   cout << "globalScope: " << globalScope << end
}
```

Variables in the **global scope** and visible to everyone

Variables visible by **foo()**

Environment	Variable or function name	Parent environment
global	globalScope	
global	foo()	
global	main()	
foo()	fooScope	global

Functions and scopes in C++

Variables in the **global scope** and visible to everyone

Variables visible by **foo()**

Undefined variables
not present in any environment
no scope (**compile time error**!)

```
#include <iostream>
using namespace std;
int globalScope = 0; //This is a global variable, visible everywhere.

void foo() {
   int fooScope = 1; //Only visible within foo function
   cout << "fooScope: " << fooScope << endl;
   cout << "localScope: " << localScope << endl;
}
int main() {
   cout << "globalScope: " << globalScope << end

    { //Any block declares a scope, even this useless
      int localScope = 3;
      cout << "localScope: " << localScope << endl
      foo();
      cout << "fooScope: " << fooScope << endl;
      int globalScope = 100; // variable hiding, very
      cout << "globalScope: " << globalScope << e
    }
   cout << "localScope: " << localScope << endl;
   cout << "globalScope: " << globalScope << end
}
```

Environment	Variable or function name	Parent environment
global	globalScope	
global	foo()	
global	main()	
foo()	fooScope	global

Functions and scopes in C++

Variables in the **global scope** and visible to everyone

Variables visible by **foo()**

Undefined variables
not present in any environment
no scope (error!)

```
#include <iostream>
using namespace std;
int globalScope = 0; //This is a global variable, visible everywhere.

void foo() {
   int fooScope = 1; //Only visible within foo function
   cout << "fooScope: " << fooScope << endl;
   cout << "localScope: " << localScope << endl;
}
int main() {
   cout << "globalScope: " << globalScope << end

    { //Any block declares a scope, even this useless
      int localScope = 3;
      cout << "localScope: " << localScope << endl
      foo();
      cout << "fooScope: " << fooScope << endl;
      int globalScope = 100; // variable hiding, very
      cout << "globalScope: " << globalScope << e
    }
   cout << "localScope: " << localScope << endl;
   cout << "globalScope: " << globalScope << end
}
```

Environment	Variable or function name	Parent environment
global	globalScope	
global	`foo()`	
global	`main()`	
`foo()`	fooScope	global
`main()`		global

Functions and scopes in C++

Variables in the **global scope** and visible to everyone

Variables visible by **foo()**

Undefined variables
not present in any environment
no scope

Variables visible in the **useless block**

```
#include <iostream>
using namespace std;
int globalScope = 0; //This is a global variable, visible everywhere.

void foo() {
   int fooScope = 1; //Only visible within foo function
   cout << "fooScope: " << fooScope << endl;
   cout << "localScope: " << localScope << endl;
}
int main() {
   cout << "globalScope: " << globalScope << endl

    { //Any block declares a scope, even this useless
      int localScope = 3;
      cout << "localScope: " << localScope << endl
      foo();
      cout << "fooScope: " << fooScope << endl;
      int globalScope = 100; // variable hiding, very
      cout << "globalScope: " << globalScope << e
    }
   cout << "localScope: " << localScope << endl;
   cout << "globalScope: " << globalScope << end
}
```

Environment	Variable or function name	Parent environment
global	globalScope	
global	`foo()`	
global	`main()`	
`foo()`	fooScope	global
`main()`		global
`Useless block`	localScope	`main()`
`Useless block`	globalScope	`main()`

Functions and ... C++

```
#include <iostream>
using namespace std;
int globalScope = 0; //This is a global variable, visible everywhere.

void foo() {
   int fooScope = 1; //Only visible within foo function
   cout << "fooScope: " << fooScope << endl;
   cout << "localScope: " << localScope << endl;
}
int main() {
   cout << "globalScope: " << globalScope << endl

    { //Any block declares a scope, even this useless
      int localScope = 3;
      cout << "localScope: " << localScope << endl
      foo();
      cout << "fooScope: " << fooScope << endl;
      int globalScope = 100; // variable hiding, very
      cout << "globa  cope: " << globalScope << e
    }
   cout <                                    endl;
   cout <                                    << end
}
```

Hidden variable!

Overridden variable name!

Variables in the **global scope** and visible to everyone

Variables visible by **foo()**

Undefined variables
not present in any environment
no scope

Variables visible in the **useless block**

Environment	Variable or function name	Parent environment
global	globalScope	
global	`foo()`	
global	`main()`	
`foo()`	fooScope	global
`main()`		global
Useless block	localScope	`main()`
Useless block	globalScope	`main()`

Functions and scopes in C++

Variables in the **global scope** and visible to everyone

Variables visible by **foo()**

Undefined variables
not present in any environment
no scope

Variables visible in the **useless block**

```
#include <iostream>
using namespace std;
int globalScope = 0; //This is a global variable, visible everywhere.

void foo() {
    int fooScope = 1; //Only visible within foo function
    cout << "fooScope: " << fooScope << endl;
    cout << "localScope: " << localScope << endl;
}
int main() {
    cout << "globalScope: " << globalScope << endl

    { //Any block declares a scope, even this useless
        int localScope = 3;
        cout << "localScope: " << localScope << endl
        foo();
        cout << "fooScope: " << fooScope << endl;
        int globalScope = 100; // variable hiding, very
        cout << "globalScope: " << globalScope << e
    }
    cout << "localScope: " << localScope << endl;
    cout << "globalScope: " << globalScope << end
}
```

Environment	Variable or function name	Parent environment
global	globalScope	
global	`foo()`	
global	`main()`	
`foo()`	fooScope	global
`main()`		global
`Useless block`	localScope	`main()`
`Useless block`	globalScope	`main()`

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 an every 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 **man▮bash**
- **`command`**
  The **boldface** typeset is usually used to identify a command or part of the string that have to be written **exactly as you read them**. In these slides I will also use the blue color, but you may not see it in the printout.
- `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 curly 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>]`
  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.

Goals and non-goals of this tutorial

- Goals:
  - Being able NOT TO PANIC when somebody gives you something you've never seen before (will happen in your entire career)
  - Being able to write a bash script.
  - Understanding the concept of **variable**. **Environment**, **binding**, **scope**.
  - Being able to search for information depending on a task one wants to achieve.
- 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

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 difference is that scripted languages **do not require compilation**.

A bash script and its components

- A **bash script** is nothing more that 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++, 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**.
- Consider the following code, a script called `getcpuinfo.sh`:

```
#!/bin/bash

# put the output of cat in the variable CPUINFO
CPUINFO=$(cat /proc/cpuinfo | head -10)

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

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.

```
CPUINFO=$( cat /proc/cpuinfo | head -10 )
```

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 was a command.

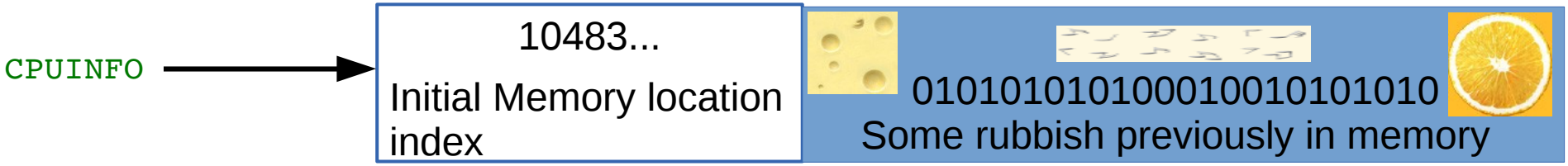
```
pflorido@tjatte:~> chmod +x getcpuinfo.sh
```

- To **run** or **execute** those in the current directory, prefix them with **./**

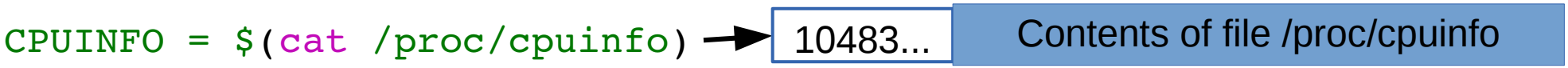
```
pflorido@tjatte:~> ./getcpuinfo.sh
processor : 0
vendor_id : GenuineIntel
cpu family : 6
model : 15
model name : Intel(R) Core(TM)2 CPU 6400 @ 2.13GHz
stepping : 6
cpu MHz : 2127.650
```

Variables, types in bash

- A **variable** is an identifier, a name, for a memory location. Its **definition** implies that the **interpreter** will find a free memory space for that variable. As in C++, this space, if not **initialized**, can contain anything.



- **Assigning** a **value** to a variable means putting such value inside that memory location.



- 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.
  - Allocation is always done dynamically depending on the assignment

Var name	Var type	Associated size	Initial tentative logical memory location pointer	value
larger	Always string	Depends on system configuration	10483392805	Contents of /proc/cpuinfo

Functions

- One can define functions to reduce complexity and increase readability

```
#!/bin/bash

# definition of a function that gets meminfo
getmeminfo(){
MEMINFO=$(cat /proc/meminfo)
}

# call to the function, it will change the environment
getmeminfo

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

- Notice the curly brackets **{ }** . These delimit a **block of code**
- The block of code above contains the **definition** of the function `getmeminfo()` that takes in input no parameters
- The `MEMINFO` variable is defined inside the definition of the function.

Environment, binding

- 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.
- 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 on the left we can see some bindings.
- When we **define** a variable, the variable name is added to the **environment**

Environment	Variable name	Starting memory index
global	PWD	48329
global	SHELL	483985
global	PATH	3412
`cpuinfo.sh`	CPUINFO	10289
`meminfo.sh`	MEMINFO	18458
`meminfo.sh`	getmeminfo()	3515

- In languages like BASH, we do not see memory indexes.
- 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)

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.

The BASH environment: export

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

1 .Run the `export` command. You'll see all the environment variables in the current bash session.

2. Create a new environment variable:

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

3. Find the variable by running `export`, or just print its content with
   `echo $MYENV1`

4. Now open another bash instance by issuing the command `bash`. Run `export`. You will find that `MYENV1` is still there.
   The environment is said to be **inherited** from the father process.

6. Open another terminal *LXTerm* and run `export`. `MYENV1` should not be there.
   There is no environment inheritance between terminal windows.
   Close the terminal and go back to the old one where `MYENV1` is defined.

Preparing for the tutorial

- Create a folder for git stuff (if you don't already have one) and change directory into it
  - `mkdir ~/git`
  - `cd ~/git`
- If you have **never** configured the git repository, clone it. A new folder MNXB01-2017 will contain its changes:
  `git clone https://github.com/floridop/MNXB01-2017.git MNXB01-2017`
- If you have the repo for HW2b configured or just cloned, cd into it:
  - `cd MNXB01-2017`
  - Add my repo as the upstream remote:
    - `git remote add upstream https://github.com/floridop/MNXB01-2017.git`
  - Pull the examples from github:
    - `git pull upstream master`
    - Save the commit message if required. You will download all the changes/pull requests that I approved during the week.
- Change directory to the tutorial folder:
  - `cd flopaganelli/Tutorial3bMaterial/bash`

Exercises

- **Exercise 3b.1:** Open geany, write and save the following code as file `getcpuinfo.sh`

```
#!/bin/bash

# put the output of cat in the variable CPUINFO
CPUINFO=$(cat /proc/cpuinfo | head -10)

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

- **Exercise 3b.2:** execute `getcpuinfo.sh` as described in slide 21.

Exercises

Exercise 3b.3:
What is the predefined PATH variable?

During the course we ran commands that did not need a **./** in front. The reason is: the directory where our code is placed is not known by the system as a place where executables are.

This list is contained in the predefined variable PATH.
Modify the first line as below, save and execute the script again:

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

Exercise 3b.4:

Enable **Debugging** to debug your script, that is, see what is doing while running, modify the first line as below:

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

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

BASH environment: scope

- Consider the bash script `envtest.sh` with the following content:

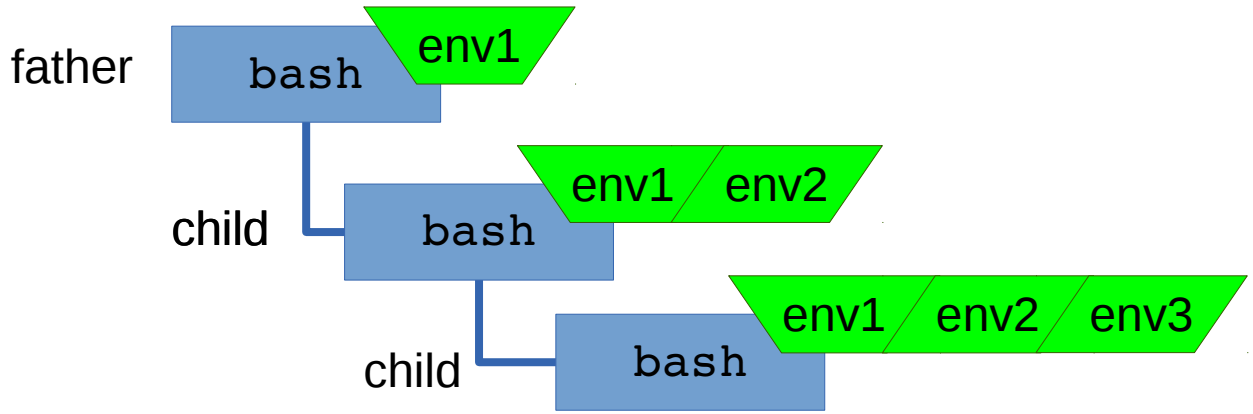
```
#!/bin/bash

# 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"'
        exit 1;
fi

# create an environment variable
MYENV2="This is my second environment variable"

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

echo "Now check if MYENV2 still exists, with the command"
echo 'echo $MYENV2'
```

BASH environment: scope

- Run it: `./envtest.sh`
- Try to run the command:

```
echo "Content of MYENV2: $MYENV2"
```

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



Importing an environment

- In bash, there is a command that allows you to copy the environment defined in a script to another script or bash instance. This command is **`source`**
- **Careful! The command also executes EVERYTHING inside the BASH script!**
- If you now try
  - **`source ./envtest.sh`**
  - **`echo`** `"Content of MYENV2: $MYENV2"`
    You'll see that the output of `export` will contain also `MYENV2`.
    `MYENV2` is now in the father bash environment.
- As a default, when you start a terminal or execute the bash command, bash sources the following files: `/etc/profile`, `~/.profile`, `~/.bash_profile`, `~/.bashrc`, and some other files, so that a set of default environment variables are defined.
- Run **`cat`** `<filename>` on the files listed above, where `<filename>` is one of the files listed and see what is in them. Ask me questions if you don't understand what you find there.

Predefined variables in scripts

- **Prefixed by the $ symbol**, they are instantiated automatically in bash at the start of the script.
- **Script 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.
- **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 Lecture 3 slides.
  - `$?`: exit code of the last executed command (0 if it ended well, any other number otherwise)
  - `$!`: PID of last command executed in background
  - ...
- **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
  - ...

Predefined variables example

```
#!/bin/bash

# predefinedvars.sh
# call with: ./predefinedvars.sh arg1 arg2 arg3
#

# print out info about arguments to this script
echo "Number of arguments: $#"
echo "Name of this script: $0"
echo "Arguments: $1 $2 $3 $4"

# print this script's Process IDentifier:
echo "PID is $$"
```

Run the script. Remember to `chmod +x predefinedvars.sh` to make it executable!

Exercise: check the output of some other predefined variable, in particular `$*` and `$@`

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 type: **`/\[\[`** `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 type: **`/\(\(`** `expression`

Conditions: Exercises

- **Exercise 3b.5**: Execute the following commands:
  - `test -e /etc`
  - `echo $?`
  - `test -e /thisfiledoesnotexist`
  - `echo $?`
  - `[ -e /etc ]`
  - `echo $?`
  - `[ -e /thisfiledoesnotexist ]`
  - `echo $?`
- **Exercise 3b.6**: Execute the following commands:
  - `[[ -e /etc ]]`
  - `echo $?`
  - `[[ -e /doesnotexist ]]`
  - `echo $?`
- **Exercise 3b.7**: Execute the following commands. Do you understand the meaning and results? If not, ask me.
  - `true`
  - `echo $?`
  - `false`
  - `echo $?`
  - `(( 0 && 0 ))`
  - `echo $?`
  - `(( 1 && 0 ))`
  - `echo $?`
  - `(( 1 && 1 ))`
  - `echo $?`

Control structures

- Enable the machine to **decide** on actions depending on certain **conditions**. (**`if..then...else..fi`**)
- 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: if ... then ... else .. fi

- The BASH syntax is as follows:

```
if <condition>; then
    <command1>;[<command2>;…]
else
    <commandA>;[<commandB>;…]
fi
```

Control structures: if ... then ... else .. fi

- `-le` = less than or equal

```
#!/bin/bash
# testif.sh
# run with: ./testif.sh arg1 arg2 arg3
#
# test that at least two 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 `exit` command

- Exit is used to terminate the program exactly where exit is called, that is, to break cycles and exit the program.
- It takes in input the return value of the process:
  - **0** for SUCCES
  - **1** for ERROR
- If you code cannot continue due to an error, you should always **`exit 1`**. Otherwise the code will continue running without the required information.
- You can test the exit value by checking the `$?` variable:

```
echo $?
```

- This works with any linux program: if there is an error, the process should exit with `$?` $\neq 0$
- Exercise: check the exit value when you input no argument or three arguments to `./testif.sh` *`[<argument1> <argument2> ...]`*

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
```

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

- Print types of files in some directory, default to the /etc directory

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


if [ "x$TARGETDIR" == "x" ]; then
     TARGETDIR='/etc'
     MESSAGE="No argument found. Listing filetypes for the /etc directory by default"
else
     MESSAGE="Scanning filetypes for the ${TARGETDIR} directory"
fi


echo "$MESSAGE"


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
```

Calling variables values in different ways

- `$VAR` returns the value contained in the variable called `VAR`.
- `${VAR}` returns the value contained in the variable called `VAR` but it makes easier to spot the boundaries of the variable name. It can be used to concatenate string values and strings, like in the previous code:

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

it shows clearly that the name of the variable is TARGETDIR

Control structures: 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`

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

- Keeps doing something as long as *<condition>* is satisfied.
- Syntax:

```
while <condition>; do
    <command1>;[<command2>;…]
done
```

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.
echo
done
exit 0
```

Control Structures: Exercises

- **Exercise 3b.8**: Change the `iftest.sh` code to complain if the user did not write at least **5** command line arguments
- **Exercise 3b.9**: Change the `listfiletypes.sh` code to list the types of files in the folder `/tmp` *by default*, that is, *if no command line argument is passed*.
- **Exercise 3b.10**: Change the `testwhile.sh` code to exit when the user writes `bye!`

Datasets

- A dataset is some digital collection, maybe a file or a set of files, that contains data we want to use.
- A dataset usually has his own **format**.
  - A format is a **set of rules** that define in a rigorous manner how the content of the dataset should be read, 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 that one needs to learn
  - 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.

Sample data file

```
<?xml version="1.0" encoding="UTF-8" ?>

<Data>
<Game>
<id>1558</id>
<GameTitle>Harvest Moon Animal Parade</GameTitle>
<ReleaseDate>11/10/2010</ReleaseDate>
<Platform>Nintendo Wii</Platform>
</Game>
<Game>
<id>32234</id>
<GameTitle>Busy Scissors</GameTitle>
<ReleaseDate>11/02/2010</ReleaseDate>
<Platform>Nintendo Wii</Platform>
</Game>
<Game>
<id>890</id>
<GameTitle>Rayman Raving Rabbids TV Party</GameTitle>
<ReleaseDate>11/18/2008</ReleaseDate>
<Platform>Nintendo Wii</Platform>
</Game>
<Game>
<id>908</id>
<GameTitle>Super Mario Galaxy 2</GameTitle>
<ReleaseDate>05/23/2010</ReleaseDate>
<Platform>Nintendo Wii</Platform>
</Game>
```

What can we say by observing this data?
Can we guess something about the structure?

Sample data file: investigation

```
<?xml version="1.0" encoding="UTF-8" ?>

<Data>
<Game>
<id>1558</id>
<GameTitle>Harvest Moon Animal Parade</GameTitl
<ReleaseDate>11/10/2010</ReleaseDate>
<Platform>Nintendo Wii</Platform>
</Game>
<Game>
<id>32234</id>
<GameTitle>Busy Scissors</GameTitle>
<ReleaseDate>11/02/2010</ReleaseDate>
<Platform>Nintendo Wii</Platform>
</Game>
<Game>
<id>890</id>
<GameTitle>Rayman Raving Rabbids TV Party</Game
<ReleaseDate>11/18/2008</ReleaseDate>
<Platform>Nintendo Wii</Platform>
</Game>
<Game>
<id>908</id>
<GameTitle>Super Mario Galaxy 2</GameTitle>
<ReleaseDate>05/23/2010</ReleaseDate>
<Platform>Nintendo Wii</Platform>
</Game>
```

What can we say by observing this data?

- It seems to be structured in some way.
- There is some metadata information at the top that might hint at some known format. Search “XML” on google?

Can we guess something about the structure?

- It seems to have opening and closing tags <tag></tag>
- The tags seems to represent a tree structure

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

Genesis of an algorithm: a top down approach

- Write a list of each main task translating the description of the problem.
- Open geany and start writing down as comments the steps to the algorithm. You can write that on paper first.
- An example of this process is the homework skeleton in git.

Homework 3b

- It's still work in progress. But when I notify you that the homework is ready, pull my repository again:
  - `git pull upstream master`
- Read the problem specification in:
  - `<MNXB01-2017 git folder>/flopaganelli/HW3b/README.md`
- Examine the skeleton file in git:
  - `<MNXB01-2017 git folder>/flopaganelli/HW3b/pokemoninfo.sh.skeleton`
- Rename the skeleton to `pokemoninfo.sh`. Complete the skeleton file with the requested lines of code.
  Test that it does what is requested! The final result should look like the files in the result folder in the github repository:
  - `<MNXB01-2017 git folder>/flopaganelli/HW3b/result`
- Commit the code to your fork in the folder **with your name** as in HW2b, in a subfolder called HW3b. For my name it would look like:
  - `<MNXB01-2017 git folder>/flopaganelli/HW3b/`
- Hint: Check the solutions of previous year assignments on the course webpage:
  http://www.hep.lu.se/courses/MNXB01/index-2016.html
  http://www.hep.lu.se/courses/MNXB01/index-2015.html

References

- Bash scripting: http://tldp.org/LDP/abs/html/