Other languages and C++
Writing scripts

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

Outline

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

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

```bash
#!/bin/bash

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

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

Anatomy of a bash script

```
#!/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"
```

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

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.

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`!

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
```

Exercises

Exercise 3b.1:
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.2:
Debugging to debug your script, that is, see what is doing while running, modify the first line as below, save and execute the script again:

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

Prepare for the tutorial
If you attended Tutorial2b (SVN)

```
# change directory into the svn local working copy
cd ~/svn/svncoursetrunk

# update the svn repository
svn update

# change directory with the username I give you on the
# piece of paper
cd the_username_I_gave_you_on_the_piece_of_paper

# Copy the code for today's tutorial from Florido's
directory
cp -r ../floridop/Tutorial3b .

# change dir into the tutorial directory
cd Tutorial3b
```

This dot is important!!!
means
“into the current directory”

Prepare for the tutorial if you didn't attend Tutorial2b (SVN)

- Run the following commands:

```
# create the svn dir
mkdir ~/svn

# change directory into the svn dir
cd ~/svn

# checkout the svn repository
svn co http://svncourse.hep.lu.se/svncourse/trunk  svncoursetrunk

# change dir into the repository you just checked out
cd svncoursetrunk

# create a directory with the username I give you on the
# piece of paper
mkdir the_username_I_gave_you_on_the_piece_of_paper

# change dir into the directory just created
cd svncoursetrunk/the_username_above/

# Copy the code for today's tutorial from Florido's directory
cp -r ../floridop/Tutorial3b .

# change dir into the tutorial directory
cd Tutorial3b
```

This dot is important!!!
means
“into the current directory”

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.

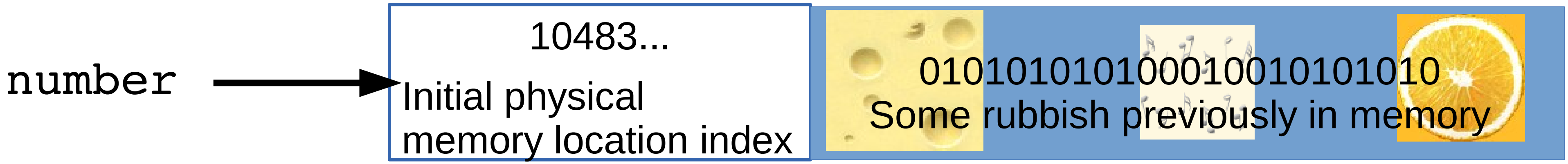
```
int number;
```

- 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, 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:



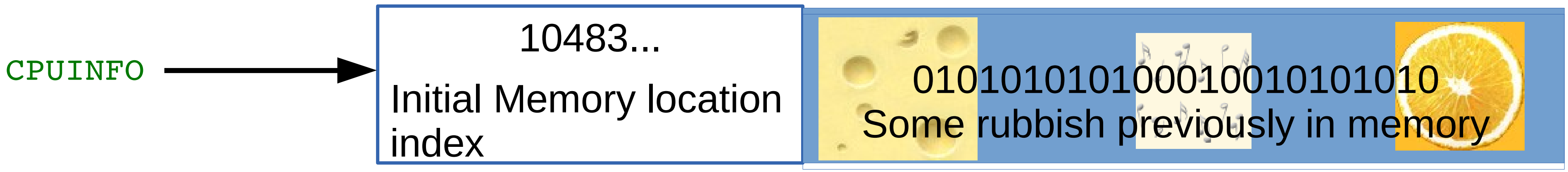
- By **assigning** a **value** to a variable, we tell the compiler what to write in the memory.

```
number = 42;
```

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

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.

```
CPUINFO = $(cat /proc/cpuinfo)
```

- In BASH, variable have no type as it is **implicitly assumed** the content is a **string**, or 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. In languages like C++ we can see them in the form of pointers.
- Binding can be:
  - **Static**, that is, decided at compilation time
  - **Dynamic**, that is, decided at execution time (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 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. **What happens if one uses the same names in two blocks???**
- 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

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. Open another bash instance by issuing the command **`bash`**. Run **`export`**. Can you find the environment variable?
The environment is said to be **inherited** from the father process.

5. Open another terminal and run **`export`**. Can you find the environment variable? There is no inheritance.

BASH environment: scope

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

```
#!/bin/bash

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

# write the content of CPUINFO to screen
echo "Content of MYENV1: $MYENV1"
echo "Content of MYENV2: $MYENV2"
```

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

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

- The father environment DOES NOT inherit from children, 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 `MYENV2` is now in the father bash environment.
- As a default, bash sources `/etc/profile` , `~/.profile` , `~/.bashrc` and some other files every time you open a terminal, so that a set of default environment variables are defined. You can `cat` these files if you're curious to see what is in them.

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 id (PID) of the script itself
  - `$?`: exit code of the last executed command (0 if it ended well, any other number otherwise)
  - `$!`: 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 PID:
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 `$*` and `$@`

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 << 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
}
```

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

Environment	Variable or function name	Parent environment
global	globalScope	

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
      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 (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 << 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

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!)

```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 << 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

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
        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()**

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 an… C++

Hidden variable!

```
#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 <…                                    …< endl;
    cout <…                                    …<< end…
}
```

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()

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`)

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
  - The double square bracket or extended test `[[ some test command ]]`. Use man bash and type: `/\[\[ expression`
    - Example: `[[ -e filename ]]`
  - The double parentheses for arithmetical expansion and logical operations `(( a && b ))`. `man bash` and type: `/\(\(expression`

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!";
else
    echo "More than 2 arguments. Good!";
fi
```

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

- Repeat something a predefinite 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 a list of files in the /etc directory

```
#!/bin/bash
# listfiles.sh
# run with: ./listfiles.sh
#
# Print the argument values
echo “Listing files in /etc”
for somefile in /etc/*; do
    echo "This is the file $somefile, with type:";
    # the file command tells you the type of a file.
    file $somefile
done
```

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
```

Exercises

- 3b.3: Change the `iftest.sh` code to complain if the user did not write at least 5 command line arguments
- 3b.4: Change the `listfiles.sh` code to list the types of files in the folder `/tmp`
- 3b.5: 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</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?

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

Automation exercise with BASH

- Description of the problem to solve:

  Write a script `checkdataset.sh` that manipulates a dataset
- The script takes in input two arguments:
  - A URL to an svn repo on the web.

```
http://svncourse.hep.lu.se/svncourse/trunk/floridop/Tutorial3b/data
```

  - A name of directory where the file and the contents of the file will be stored

Genesis of an algorithm: a top down approach

- Write a list of each main task translating what I wrote in the description. We can brainstorm it in the class before proceeding.
- Open a new .sh file with geany
- Write down the header and start writing down as comments the steps to the algorithm. You can write that on paper first.
- An example is placed in svn as `floridop/Tutorial3a/homework/checkdataset.sh.skeleton`

Inspecting the dataset

1. Download it from svn with the command:

```
svn co http://svncourse.hep.lu.se/svncourse/trunk/floridop/Tutorial3b/data dataset
```

2. List the content of the dataset directory

```
ls -ltrah dataset
```

3. Open the file with geany

```
geany dataset/nintendowiigamesprettyprinted.xml &
```

Homework 3b

- Download the skeleton file from svn

```
http://svncourse.hep.lu.se/svncourse/trunk/floridop/Tutorial3b/homework/checkdata
set.sh.skeleton
```

- Complete the skeleton file with the requested lines of code.
- Upload the code to training.lunarc.lu.se
- The final result should look like the files at the url:

```
http://svncourse.hep.lu.se/svncourse/trunk/floridop/Tutorial3b/homework/sampleresult/
```

References

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