ICTP DP Linux Basic Course - Programming

ESP Students - First Semester

Graziano Giuliani
ggiulian@ictp.it

The Abdus Salam International Centre for Theoretical Physics

ICTP Diploma Program
September 5, 2023

Course Outline [1]

Daily program

- **UNIX/Linux**
- **Programming on Linux**
  1. Programming Language
  2. Compile using Fortran language
  3. Python 1-2-3
- Text file manipulation
- Basic BASH and Python

Slides:

`http://tinyurl.com/2jsvfbd6`

or the LaTeX source on GitHub:

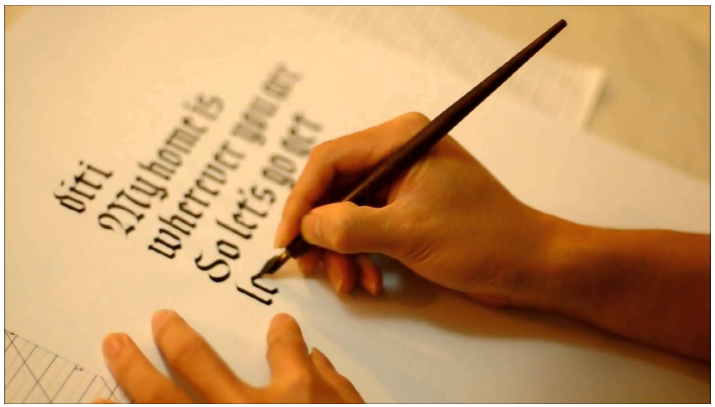
`https://github.com/graziano-giuliani/LinuxBasics`

[1] Course created in 2019 with Adriano Angelone, now LPTMC-FR

Computer Programming

Let the computer do the hard work!

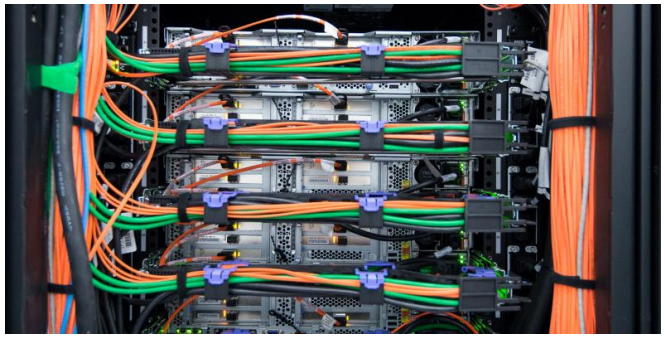
- What is the computer good for?
  - Work 24/7 if power available
  - Perform repetitive and boring task
  - Do large computation without error
  - Store reliably large amount of data
- How to make it work for you?
  - The computer thinks binary
  - Translation required
  - The computer translate
  - Exact grammar and syntax
  - No errors allowed



Scientific Programming

You can do Science with a computer!

- Tools and libraries for data acquisition
- Modelling and numerical algorithms
- Data storage and visualization



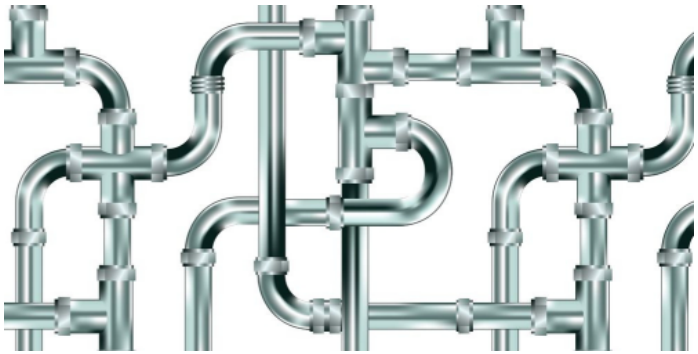
It can get to such complexities that a whole new Science has emerged:

Computer Science : study of computers and computational systems

Writing your own program

The UNIX way

Use the building blocs of existing programs and create a complex **pipeline** of stages to reach the desired processing



- **Pros** : No programming in the general sense involved, just carefully examination of the input and output of existing system programs to create the required processing.
- **Cons** : Limited by the possible processing allowed by system programs, generally related to text file manipulation, non portable across different systems

Example Shell Scripting:

```
#!/bin/bash

ls -al | grep ${USER} | tr -s ' ' | cut -d " " -f 5 > $1
```

Writing your own program

Interpreted languages

Use generic **scripting language** interpreters which can more flexibly allow runtime evaluation of a processing

- **Pros** : More flexible, eventually the shell itself can be used, can use specialized libraries for compute intensive tasks, rapid prototyping
- **Cons** : Need to learn a programming language, not as fast as a system binary can be.

Example:

- Python language
- Julia language
- R statistical language

Writing your own program

Compiled languages

Use a low level **programming language** which is parsed by a program called *compiler* to create a system binary program

- **Pros** : Fast execution time, tailored processing to the problem to solve
- **Cons** : Need to learn a programming language, not as flexible as a scripting language, may require writing code even for very simple and common tasks best approached by generic system programs.

Example:

- Fortran Programming Language
- C/C++ Programming Language

Writing your own program

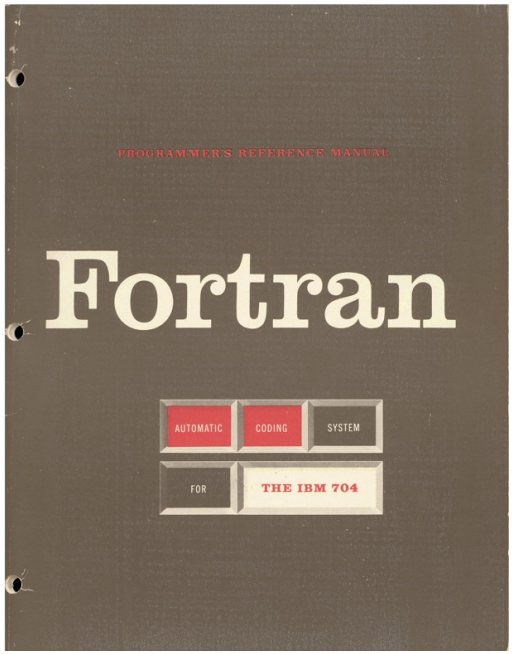
Which language is THE best

There is no a **win all** language.

- Do not use a sledgehammer to crack walnuts
- Select the language keeping in mind the problem to solve
- Always start from the simplest solution
- Invest in learning a new tool only if needed
- Consider the problem at hand and split it in simple tasks
- Use the best possible tool for each task
- Identify performance bottlenecks and solve those

Fortran Language

Programming like in the 1950



- Fortran programs are witten as source text files:

```
program count
  integer :: i
  do i = 1, 10
    print *, i
  end do
end program count
```

- A **compiler program** parses source files and create binary object files:

```
gfortran -o myprog myprog.f90
```

- Objects are linked with other objects or libraries to create executables:

```
0000000 457f 464c 0102 0001 0000 0000 0000 0000
0000010 0003 003e 0001 0000 06f0 0000 0000 0000
0000020 0040 0000 0000 0000 1a78 0000 0000 0000
0000030 0000 0000 0040 0038 0009 0040 001d 001c
0000040 0006 0000 0004 0000 0040 0000 0000 0000
```

Exercise

Write and compile Fortran 'Hello World'

- Create a text file `hello.f90` with the below content:

```
program hello
  implicit none
  integer , parameter :: stdout = 6
  character(len=*), parameter :: message = 'Helo World'
  character(len=1), parameter :: mark = '!'
  write(stdout,'(a,a)') message, mark
end program hello
```

- Compile the program into an executable file:
  `gfortran -o hello hello.f90`
- Execute the executable file created:
  `./hello`

Have you noticed?

No error is allowed in the code: Fortran Language has very strict syntax.
But you can misspell the message!
Why we had to run the program with `./hello`?

File Permissions

How to check file permissions

See permissions: `ls -l`

```
~/example_dir » ls
file_1  file_2  file_3  file_4
-------------------------------------------------
~/example_dir » ls -l
total 0
-rw-r--r-- 1 nemesis3 users 0 Sep 25 00:58 file_1
-r-xr--r-- 1 nemesis3 users 0 Sep 25 00:56 file_2
-rw-rw-rw- 1 nemesis3 users 0 Sep 25 00:56 file_3
-rwxr-xr-x 1 nemesis3 users 0 Sep 25 00:56 file_4
```

Check this

What do you see running `ls -l` in the directory where you have compiled the `hello` program and code?
Can you spot the executable files looking at the permission bits?
Try running `ls -al`. Can you tell which are the directories from the permission bits?

Executable files

System programs and user programs

- An executable file is a file with execute permission bit on
- The system programs (`ls`, `mkdir`, `cd`, `gfortran`, etc.) are files with execute permission bit set that are stored in predefined directories (`/bin`, `/usr/bin`, `/usr/local/bin`) that are in the system `PATH` for searching programs
- User executable are not in the `PATH` and to run them the path to reach the executable, absolute or relative, must be provided
- `PATH` is an environment variable. More on those next days.

Check the system programs

Try searching the program `ls`.

```
ls -l /bin/ls
```

Where is `cd`? And `gfortran`?

Compiler flags

Optional arguments for Fortran programming

The compiler is a program and accepts command line arguments. Above we have used the `-o` option to specify the name of the output.

Useful gfortran options

- `-o program` output program name (default a.out)
- `-Ofast` Optimize program for fast execution time
- `-O0` Remove all optimization (for debugging)
- `-g` include debugging information
- `-Wall` Enables commonly used warning options pertaining to usage recommend avoiding and that are easy to avoid
- `-pedantic` check program for Fortran standard conformance
- `-fcheck=all` perform all available run-time checks
- `-fbacktrace` print the whole trace of the error

Exercise

Debug a Fortran program

- Create a text file `error.f90` with the below content:

```
program error
  implicit none
  integer, parameter :: im = 2
  integer, dimension(im,im) :: matrix
  integer :: j
  do j = 1 , 2*im
    do i = 1 , 2*im
      matrix(i,j) = i*j
      print*, matrix(i,j)
    end do
  end do
end program error
```

- Compile and correct the error
- Run the program and check the runtime error
- Compile in debug: `-O0 -g -Wall -pedantic -fcheck=all -fbacktrace`
- Run the program, check the error and correct the program
- Recompile an optimized executable

File permissions

RWX bits

There are three basic attributes for plain file permissions:

- read `r`
- write `w`
- execute `x`

They mean what you would expect. There are three classes of users:

- owner `u`
- group `g`
- other `o`

For each of the three classes you have three possible attributes to set.

For directories:

- read : you can list the content
- write : you can create/remove files inside
- execute : you can access it and its content

File Permissions

Changing file permissions using `chmod`

Change permissions: `chmod`

```
chmod <who><+/-><what> <file>
```

- `<who>`:
  `u` (user),
  `g` (group),
  `o` (other),
  `a` (everybody)
- `<+/->`: `+` to add, `-` to remove
- `what`: `r,w,x` as above

```
~/example_dir » ls -l
total 0
-rw-r--r-- 1 nemesis3 users 0 Sep 25 01:03 file_1
-rw-r--r-- 1 nemesis3 users 0 Sep 25 01:03 file_2
-rw-r--r-- 1 nemesis3 users 0 Sep 25 01:03 file_3
-rw-r--r-- 1 nemesis3 users 0 Sep 25 01:03 file_4
-------------------------------------------------
~/example_dir » chmod u+x file_2
-------------------------------------------------
~/example_dir » chmod u-w file_2
-------------------------------------------------
~/example_dir » ls -l
total 0
-rw-r--r-- 1 nemesis3 users 0 Sep 25 01:03 file_1
-r-xr--r-- 1 nemesis3 users 0 Sep 25 01:03 file_2
-rw-r--r-- 1 nemesis3 users 0 Sep 25 01:03 file_3
-rw-r--r-- 1 nemesis3 users 0 Sep 25 01:03 file_4
-------------------------------------------------
~/example_dir » chmod a+w file_3
-------------------------------------------------
~/example_dir » ls -l
total 0
-rw-r--r-- 1 nemesis3 users 0 Sep 25 01:03 file_1
-r-xr--r-- 1 nemesis3 users 0 Sep 25 01:03 file_2
-rw-rw-rw- 1 nemesis3 users 0 Sep 25 01:03 file_3
-rw-r--r-- 1 nemesis3 users 0 Sep 25 01:03 file_4
-------------------------------------------------
~/example_dir » chmod a+x file_4
-------------------------------------------------
~/example_dir » ls -l
total 0
-rw-r--r-- 1 nemesis3 users 0 Sep 25 01:03 file_1
-r-xr--r-- 1 nemesis3 users 0 Sep 25 01:03 file_2
-rw-rw-rw- 1 nemesis3 users 0 Sep 25 01:03 file_3
-rwxr-xr-x 1 nemesis3 users 0 Sep 25 01:03 file_4
```

Scripts and programs

Creating a shell script executable

Let us create an 'hello world' program in shell and execute it.

Shell script

Edit a text file `hello.sh` and write inside the following:

```
#!/bin/bash
echo Hello world
```

Can you directly execute the script as we have done with `hello`?
Try `./hello.sh`

Use `chmod`

```
chmod u+x hello.sh
```

Can you now execute the script?

Notes on the script

Missing bits

sha-banging magic number

The control sequence `#!` at the script start is a so called hash bang, or shabang or shebang. The purpose is to act as a magic sequence. The program name that follows is the program that must be called to execute what is following.

```
#!/bin/bash
```

```
#!/usr/bin/env python3
```

```
#!/usr/bin/perl
```

echo

The program `echo` is `/bin/echo` and it is used to display on the screen any provided line following the program name.
Look at the man page for it options.

Python language

Running the python interpreter

Python is an interpreted language. Any line of code goes through the interpreter in a **Read-Eval-Print Loop** or **REPL**.

Install the interpreter

```
ictp-install python3 ipython3 python3-ipython
ictp-install python3-numpy python3-matplotlib
```

Start the interpreter

```
ipython3
```

Test the interpreter

```
64*2
```

Start the interpreter

```
ipython3
```

Copy line by line the following:

```
from datetime import datetime
import numpy as np
import matplotlib.pyplot as plt

my_data = np.genfromtxt('first.csv',delimiter=',',dtype=None)
precip = np.fromiter((x[3] for x in my_data),float)
dt = list(datetime(x[0],x[1],x[2]) for x in my_data)
timeaxis = np.array(list(np.datetime64(x) for x in dt))
plt.plot(timeaxis,precip)
plt.gcf().autofmt_xdate()
plt.show( )
```

Python language

From interactive to program

Write the following in the file `plot.py`

```
#!/usr/bin/env python3
from datetime import datetime
import numpy as np
import matplotlib.pyplot as plt
my_data = np.genfromtxt('first.csv',delimiter=',',dtype=None)
precip = np.fromiter((x[3] for x in my_data),float)
dt = list(datetime(x[0],x[1],x[2]) for x in my_data)
timeaxis = np.array(list(np.datetime64(x) for x in dt))
plt.plot(timeaxis,precip)
plt.gcf().autofmt_xdate()
plt.show( )
```

What is missing?

There is still one passage missing to be able to run the program as `./plot.py`. Can you tell me what is missing?