Programmazione C++ per la Fisica

Ramona Lea
Università degli studi di Trieste
Laurea Magistrale in Fisica
A.A. 2018/2019

Mail : ramona.lea@ts.infn.it

www.ts.infn.it/~lea/cpp2018.html

Moodle: https://moodle2.units.it/course/view.php?id=4172
Corsi 2018/2019- 739SM- PROGRAMMAZIONE C++ PER LA FISICA 2018

References

- Slides and other material:
  - http://www.ts.infn.it/~lea/cpp2018.html
  - Moodle UniTs
- On line resources:
  - ◆http://www.learncpp.com
  - ◆http://www.cplusplus.com
  - ◆http://root.cern.ch
- Book
  - “Programming with C++” John R. Hubbard, Schaum’s outlines
  - “C++ How to Program- Fourth Edition”, by H. M. Deitel, P. J. Deitel, Prentice Hall, New Jersey, 2003, ISBN: 0-13-038474.
  - “The C++ programming language” Bjarne Stroustrup, Addison-Wesley Professional, 3 edition (1997), ISBN: 978-0201889543
  - “Scientific and Engineering C++: An Introduction with Advanced Techniques and Examples”, John J. Barton, Lee R. Nackam, Addison Wesley (1994), ISBN: 978-0201533934

Timetable and final examination

Place: here: Aula T21

- Timetable:

  (almost) each Friday from 14.00 to ~17.30

  - Lessons that will be missed:
    - 29/03/19 : I’ll be away for a conference
    - 19/04/19 : Easter break
    - 26/04/19 : Physics Department closed
- Lectures structure: (a bit of) theory and (a lot of) programming will be mixed during the afternoon
- Examination, two steps:
  – “written part” coding an analysis program (at home)
  – “oral part”: running and discussion of the code

Introduction

Computers in Physics

- Remote control, slow control
- Data acquisition
- Data storage
- Data reduction (from raw data to observable)
- Data analysis
- Detectors simulation
- Data and information exchange
- Info's research
- Publications

Linux

- Linux is an Operating System
  - Linux is the kernel code
  - Linux is POSIX (Portable Operating System Interface) compliant, is a Unix standardization
- Other OS are: Windows, OS-X, Android,…
- Linux kernel + additional software to interface to humans for any needed task
- Many flavors including:
  - –Debian
  - –Slackware
  - –RedHat (Fedora Core, Enterprise)
  - –Suse
  - –Mandrake
  - –Gentoo
  - –Ubuntu
  - –Scientific Linux

X-interface

XFree86 is the open source X-windows manager in most (all?) distributions

- ... then we need a “windows manager”
  - twm
  - fwm
  - ...
- ... or even better a “desktop manager”
  - ICE
  - KDE
  - GNOME
  - ...

Basic interface, the shell

- OK, with X windows we can do many things but the basic interface to the OS is the shell
- The shell is a command line interpreter, it reads the user input and execute the given command(s)
- The “command prompt” is the line where the user writes command
- The shell (usually) runs inside a terminal window
- Most common shells:
  - sh: Bourne Shell
  - csh: C Shell
  - ksh: Korn Shell
  - tcsh: Enhanced C Shell
  - bash: Bourne Again Shell

The file system

- The file system represents the way information are stored on the mass memory
- Where are data? in a hierarchical organization of directories and files, a “tree”
- The hierarchical tree develop from a “root”, the name of the “root” is a single character: /
- Directories are files which contain other files and directories; directories are files which contain the infos of their content, the “filenames”
- The “filenames” are the names of the files in a directory. “/” is not allowed as character in the filenames

The Linux file system

```
/-
 |- /bin-- all basics executable
 |- /boot-- files needed to boot the system
 |- /home -- users home directories
 |- /usr -- everything needed by a user
 |- /usr/local binaries, libraries, include files etc. etc. etc.
 |- /usr/bin
 |- /usr/lib
 |- /usr/include
 |- /include -- system headers files
 |- /lib -- system libraries and driver modules
 |- /etc -- system configuration files
 |-..........
```

Basic shell commands and scripting

http://tldp.org/HOWTO/Bash-Prog-Intro-HOWTO.html

Bash

- Bash is an acrimonious for Bourne-Again shell from the classic Bourne shell
- Bash is the “de facto” standard for shell scripting on many UNIX flavors
- Possible things you can do:
    - I/O Redirection
    - Pipe
    - Expansion
    - Define variables
    - cp, mv, ls, mkdir, cd, pwd
    - Write some script to let the computer do boring operations

Learning bash

Some basic commands:

- create a new directory (**`mkdir`**)
- enter the directory (**`cd`**)
- use a text editor (**`vi`**, **`nano`**, **`emacs`**…) to create a script that will print out on the STDOUT a sentence (any)
- run the script and redirect the STDOUT to a file (**>** or **>>**)
- run again and redirect to a file with a different name
- change the filenames adding the string "_test.txt" (pippo.txt-> pippo.txt_text.txt , …) using a loop in a single line command (bash commands: **`for`**, **`do`**, **`done`**, **`mv`**,…)

How to...

- ... move in the filesystem?

  command “`cd`”

- ... list directory content?

  command “`ls`”

  `./` means “this directory”

  `../` means “the upper directory”

- get help for a command: usually

  `command –h (command --help)`

  or

  `man command` (“man” stands for manual)

- ... look inside a file?

```
cat filename
less filename
more filename
```

Some other commands

- **`cp`** - copy files and directories

```
cp [OPTION]... [-T] SOURCE DEST
cp [OPTION]... SOURCE... DIRECTORY
-f, -i, -r
```

- **`mv`**- move (rename) files

```
mv [OPTION]... [-T] SOURCE DEST
mv [OPTION]... SOURCE... DIRECTORY
-f, -i
```

- **`ls`** - list directory contents

```
ls [OPTION]... [FILE]...
-a, -d, -h, -l
```

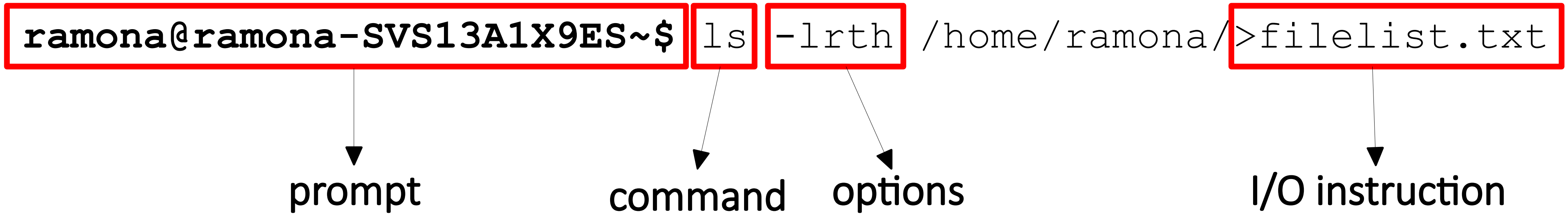
- **`mkdir`** - make directories

```
mkdir [OPTION] DIRECTORY...
-p
```

- **`pwd`** - print name of current/working directory

```
pwd [OPTION]
```

Basics bash commands



```
ramona@ramona-SVS13A1X9ES~$ ls -lrth /home/ramona/>filelist.txt
```

Under unix any files can be:

- r – read
- w – written
- x – executed

by:

- o – others, anybody, the world
- g – group
- u – user only, owner of the file

http://tldp.org/HOWTO/Bash-Prog-Intro-HOWTO.html

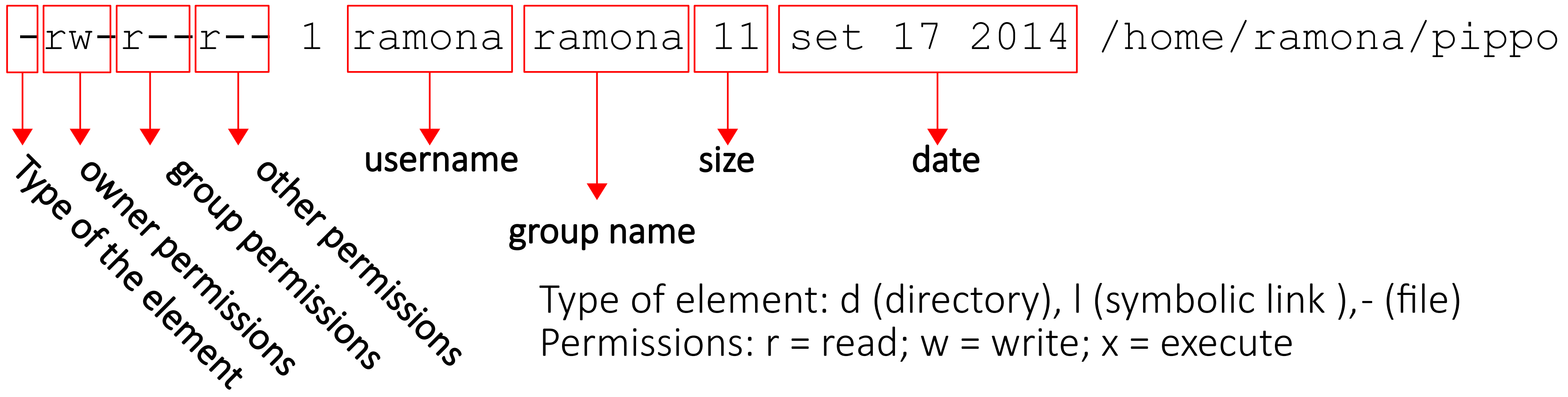
Basics bash commands

```
ramona@ramona-SVS13A1X9ES ~ $ ls -lrth ~/pippo

-rw-r--r-- 1 ramona ramona 11 set 17 2014 /home/ramona/pippo
```

Basics bash commands

```
ramona@ramona-SVS13A1X9ES ~ $ ls -lrth ~/pippo
```

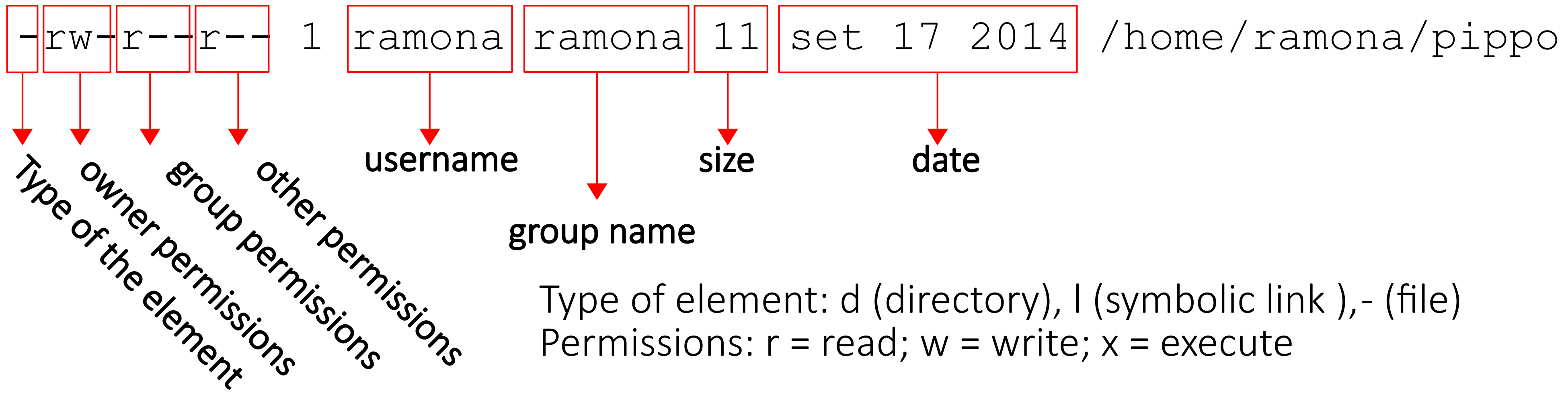


```
-rw-r--r-- 1 ramona ramona 11 set 17 2014 /home/ramona/pippo
```

Type of element: d (directory), l (symbolic link),- (file)
Permissions: r = read; w = write; x = execute

Basics bash commands

```
ramona@ramona-SVS13A1X9ES ~ $ ls -lrth ~/pippo
```



```
-rw-r--r-- 1 ramona ramona 11 set 17 2014 /home/ramona/pippo
```

Type of element: d (directory), l (symbolic link),- (file)
Permissions: r = read; w = write; x = execute

How to change permissions : **chmod**

Symbolic Notation

```
chmod a=rwx namefile
```

a (all)
u (owner)
g (group)
o (other users)

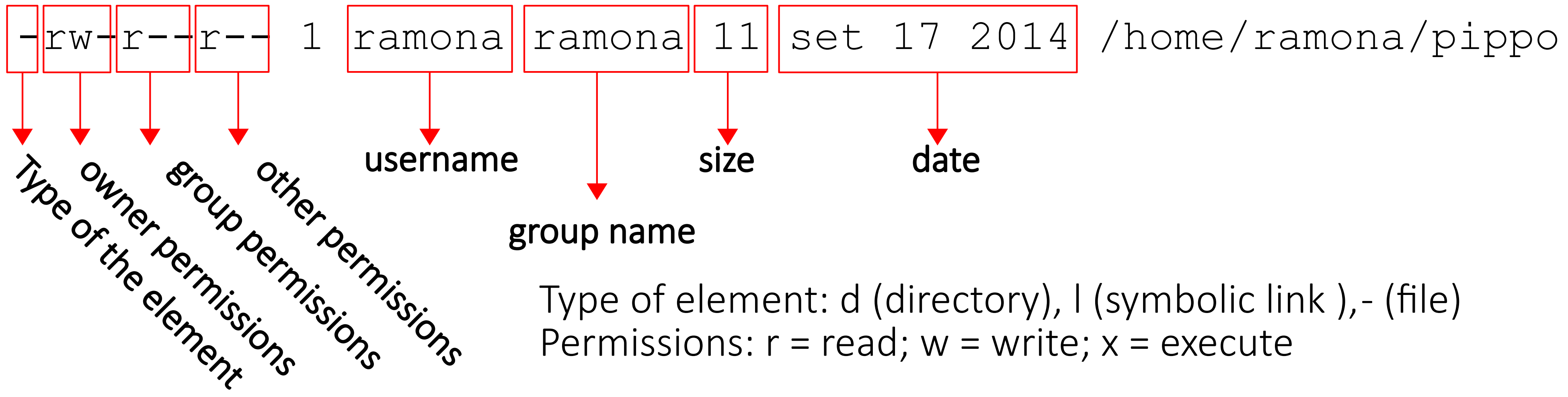
Octal Notation

```
chmod 777 namefile
```

7 corresponds to rwx	3 corresponds to wx
6 corresponds to rw	2 corresponds to w
5 corresponds to rx	1 corresponds to x
4 corresponds to r	0 access denied

Basics bash commands

```
ramona@ramona-SVS13A1X9ES ~ $ ls -lrth ~/pippo
```



```
-rw-r--r-- 1 ramona ramona 11 set 17 2014 /home/ramona/pippo
```

Type of element: d (directory), l (symbolic link),- (file)
Permissions: r = read; w = write; x = execute

How to change ownership: **chown**

```
   chown owername:groupname filename
   chown owername filename
```

I/O Redirection

There are three types of file descriptors:

1)standard input: stdin

2)standard output: stdout (1)

3)standard error: stderr (2)

```
command < file.in
```

The input is taken from file.in instead of from stdin

```
command > file.out
```

The output is redirected from stdout to file.out (file.out, if present, is overwritten)

```
command >> file.out
```

The output is redirected from stdout to file.out (if file.out is present, the output is added at the end of the file)

Pipe

- `command1 | command2`

  command1 is executed and the stdout of command1 is used as stdin of command2

- Multiple commands can be chained

```
cat *.txt | sort | uniq > result-file
# Sorts the output of all the .txt files and
deletes duplicate lines
# finally saves results to "result-file"
```

Executables

- Any executables is run calling its name:

```
/home/lea/test.exe
/usr/local/bin/mozilla
./my_print
```

- So why do we call "`ls`, `cat`, `etc.`" and not "`/bin/ls`, `/bin/cat`, `/bin/etc.`"?

- bash uses **ENVIRONMENTAL VARIABLES** to make life easier

Bash environmental variables

- If full path is not given, any executables is searched in the directories listed in the environmental variable called "`PATH`".
- How to print an environmental variable?

```
ramona@ramona-HP-ZBook-14u-G5:~$ echo $PATH
```

```
/usr/local/sbin:/usr/local/bin:/usr/sbin:/
usr/bin:/sbin:/bin:/usr/games:/usr/local/
games
```

- Directories are separated by colons ":"
- How to add a directory?

```
export PATH=/home/ramona/bin:$PATH
```

Bash environmental variables

- Libraries contain code and data that provide services to independent programs. This encourages the sharing and changing of code and data in a modular fashion, and eases the distribution of the code and data. Library files are not executable programs.
- A shared library or shared object is a file that is intended to be shared by executable files and further shared objects files. Modules used by a program are loaded from individual shared objects into memory at load time load or run time
- Shared libraries are searched at load time or run time in the directories listed in the environmental variable called "`LD_LIBRARY_PATH`".
- How to print shared libraries?

```
ramona@ramona-SVS13A1X9ES ~ $ echo $LD_LIBRARY_PATH
```

- directories are separated by colons ":"
- How to add a directory?

```
export LD_LIBRARY_PATH=/home/ramona/lib:$LD_LIBRARY_PATH
```

Bash scripting

- **Bash script**: list of bash commands written in a text file usually having suffix "`.sh`"
- Scripts can be run with:
  - `prompt> source script.sh`
  - `prompt>.script.sh` (within current shell session)
  - `prompt>./script.sh` (but first you must make script.sh executable) it opens a new session

Bash scripting

- **Bash script**: list of bash commands written in a text file usually having suffix "`.sh`"
- Scripts can be run with:
  - `prompt> source script.sh`
  - `prompt>.script.sh` (within current shell session)
  - `prompt>./script.sh` (but first you must make script.sh executable) it opens a new session
- **Minimal bash script**:
  - edit a new file, let's say test.sh and write:

    ```
    #!/bin/bash
    echo "This is test file!"
    ```
  - run it with in the three possible ways.

Bash scripting

- Now change it to:

```
#!/bin/bash
cd /tmp
pwd
echo “This is test file!”
```

- run it with in the three possible ways.
- Now create, inside the `$HOME/test` folder the file `test.sh`

```
#!/bin/bash
echo “This is test2 file!”
```

- Make it executable:

```
chmod +x test.sh
```

- In the shell type:

```
export PATH=$PATH:~/test
```

- Now you should be able to run your script just typing

```
|prompt> test.sh
```

Variables

The name of the variable is the container of its value, the memorized data. The reference to this value is called “substitution”

```
bash ~ $ variabile=23
bash ~ $ echo variabile
variabile
bash ~ $ echo $variabile
23
```

Some particular variables:

```
bash ~ $ $RANDOM
```

Contains a pseudo-casual number

In a script:

`$0, $1, $2,...`	Positional parameters
`$@`	All the positional parameters (but `$0`)
`$#`	Numbers of Positional parameters (but `$0`)

Expansions

After the "words recognition", the Bash interpreter does the *expansions*

Order of expansions:

> brace expansion, tilde expansion, parameter, variable and arithmetic expansion and command substitution (done in a left-to-right fashion), word splitting, and pathname expansion.

Only brace expansion, word splitting, and pathname expansion can change the number of words of the expansions; the other expands in a single world

Expansions

- Brace expansion

```
bash ~ $ echo a{d,c,b}e
ade ace abe


bash ~ $ echo {x,y,z}
x y z
```

- Expansion can be nested:

```
bash ~ $ echo A{b{1,2,4},c,g}FFF
Ab1FFF Ab2FFF Ab4FFF AcFFF AgFFF
```

Special Characters

- ? any character (one and only one character)

```
bash ~ $ls .b?shrc
bash ~ $ls .b?shr?c
```

- * any character, even none

```
bash ~ $ls .b*
bash ~ $ls .b*shrc
bash ~ $ls .b*rc
bash ~ $ls .b*r*c
```

Loop

```
for arg in [list]
do
  command(s)...
done
```

Examples:

```
  for planet in Mercury Venus Earth ;
  do echo $planet ; done

  for i in $(seq 1 100); do echo $i;
  done
```

Some other commands

`grep` – print lines matching a pattern

```
grep [OPTIONS] PATTERN [FILE...]
-i Ignore case distinctions in both the PATTERN and the input files.
-v Invert the sense of matching, to select non-matching lines.
```

Other options:

```
man grep or grep –help
```

`wc` – print newline, word, and byte counts for each file

```
wc [OPTION]... [FILE]…
-c print the byte counts
-l print the newline counts
-w print the word counts
```

`tr` – translate or delete characters

```
tr [OPTION]... SET1 [SET2]
SETs are specified as strings of characters.
-d delete characters in SET1, do not translate
-s replace each input sequence of a repeated character that is listed
in SET1 with a single occurrence of that character
```

Command `grep`

Example: How to filter out from the command output or from a physical file the lines containing the certain pattern?

Suppose you have this file:

```
TEST Test test 11test test22
test TEST Test 11test test22
TeST1 TEST1 TESt1 TEST1 TEST1
test TEST Test 11test test
TeST2 TEST2 TEsT2 TEST2 tEST2
```

With grep command you can do lots of stuff like:

```
# print all lines in the file containing pattern "test"
grep "test" grepExample.txt
# print all lines in the file containing pattern "test", and the numbers of those lines:
grep -n "test" grepExample.txt
# inverse search: print all lines in the file which does NOT contain the pattern "test"
grep -v "test" grepExample.txt
# case insensitive search: print all lines in the file which contain the pattern "test" or "TEST" or
"tEsT", etc.
grep -i "test" grepExample.txt
# beginning of the line: print all lines in the file which contain the pattern "test" ONLY at the
beginning of the line
grep "^test" grepExample.txt # => use anchor ^
# end of the line: print all lines in the file which contain the pattern "test" ONLY at the end of
the line
grep "test$" grepExample.txt # => use anchor $
```

Command grep

```
# word begins with the pattern: print all lines in the file which contain the word which begins with the
pattern "test"
grep "\<test" grepExample.txt

# word ends with the pattern: print all lines in the file which contain the word which ends with the
pattern "test"
grep "test\>" grepExample.txt

# exact match: print all lines in the file which contain the word which is exactly the same to the
pattern "test"
grep "\<test\>" grepExample.txt

# OR: print all lines in the file which contain either the pattern "11test" or "test22"
grep "11test\|test22" grepExample.txt # => within grep, OR is represented with \|

# AND: print all lines in the file which contain both the pattern "11test" and "test22"
grep "11test" grepExample.txt | grep "test22" # => within grep, there is no built-in AND operator, but
piping saves the day

# quiet grep: if you are just interested if a file contains the certain pattern, without actual printout
grep -q "11test" grepExample.txt && echo "yes, file contains pattern 11test" || echo "no, file doesn't
contain pattern 11test"

# filter out of lines with the pattern "test" to another file:
grep "test" grepExample.txt 1> grepOutput.log
cat grepOutput.log
```

Command **awk**

Example: How to select specified fields (columns), either from a file or from a command output?

Try to compare the output of

```
date
date | awk '{print $4}'
date | awk '{print $6}'

# To get the entry from the last column, you can also use:
date | awk '{print $NF}'

# The default field separator in awk is blank character. If you
want to use another field separator, for instance ":", use:

date | awk 'BEGIN {FS=":"}{print $<column-number>}'
```

Processes

Two most frequently used commands to handle running processes are `top` and `ps`

The `top` program provides a dynamic real-time view of a running system. It can display system summary information as well as a list of processes or threads currently being managed by the Linux kernel. The types of system summary information shown and the types, order and size of information displayed for processes are all user configurable and that configuration can be made persistent across restarts.

Quit top: type `q`

The ps displays information about a selection of the active processes. If you want a repetitive update of the selection and the displayed information, use top(1) instead.

To see every process on the system using standard syntax:

```
ps -ef
```

If you are not the only user of the computer don't be afraid to use ps with grep (with a pipe)

```
ps – ef | grep lea
```

If you want to kill this particular job, you can do

```
kill process_number
```

If the program do not stop use rude force

```
kill -9 process_number
```

Command `find`

How to find files or directories on your local hardisk?

Usage: `find <where> <what> <optionally-do-something-on-what-you-have-found>`

Example: Find all files in the specified directory

```
find <path-to-directory> -type f
```

Example: Find all files with an extension ".pdf" in the specified directory

```
find <path-to-directory> -type f -name "*.pdf"
```

Example: Find all files with an extension ".pdf" larger than 10k in the specified directory

```
find <path-to-directory> -type f -name "*.pdf" -size
+10k
```

Editors

File (text) editors

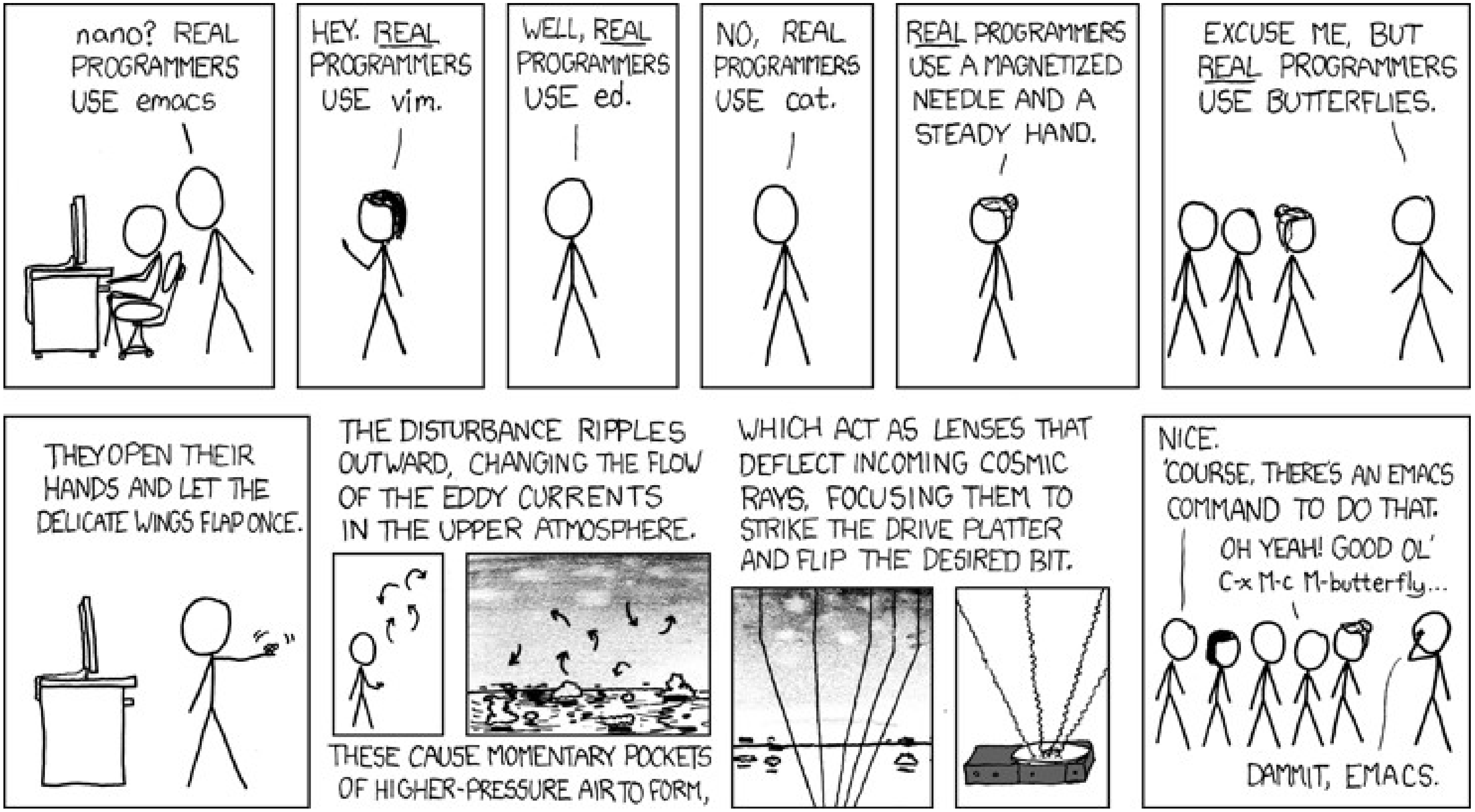
- xemacs
- emacs
- eclipse
- vi/vim
- nano/pico
- office
- word ... ...
- more than simple editors
- http://en.wikipedia.org/wiki/Comparison_of_text_editors

  in principle any editor is good but...

Editors

- Choose an editor which is good for coding (nedit or gedit ok, there exists also editors with embedded C++ compiler, I like emacs, even if is not user-friendly)
- A typical editor designed for coding has a few features that make programming much easier, including:
  - **Line numbering**. Line numbering is useful when the compiler gives us an error. A typical compiler error will state “error, line 64″. Without an editor that shows line numbers, finding line 64 can be a real hassle.
  - **Syntax highlighting and coloring**. Syntax highlighting and coloring changes the color of various parts of your program to make it easier to see the overall structure of your program.
  - **An unambiguous font**. Non-programming fonts often make it hard to distinguish between the number 0 and the letter O, or between the number 1, the letter l (lower case L), and the letter I (upper case i). A good programming font will differentiate these symbols in order to ensure one isn’t accidentally used in place of the other.
  - **Indentation capabilities**. C/C++ do not care about spaces and code text formatting, but humans and source code management programs do!!

Editors war


nano? REAL PROGRAMMERS USE emacs
HEY. REAL PROGRAMMERS USE vim.
WELL, REAL PROGRAMMERS USE ed.
NO, REAL PROGRAMMERS USE cat.
REAL PROGRAMMERS USE A MAGNETIZED NEEDLE AND A STEADY HAND.
EXCUSE ME, BUT REAL PROGRAMMERS USE BUTTERFLIES.
THEY OPEN THEIR HANDS AND LET THE DELICATE WINGS FLAP ONCE.
THE DISTURBANCE RIPPLES OUTWARD, CHANGING THE FLOW OF THE EDDY CURRENTS IN THE UPPER ATMOSPHERE.
THESE CAUSE MOMENTARY POCKETS OF HIGHER-PRESSURE AIR TO FORM,
WHICH ACT AS LENSES THAT DEFLECT INCOMING COSMIC RAYS, FOCUSING THEM TO STRIKE THE DRIVE PLATTER AND FLIP THE DESIRED BIT.
NICE.
'COURSE, THERE'S AN EMACS COMMAND TO DO THAT.
OH YEAH! GOOD OL' C-x M-c M-butterfly...
DAMMIT, EMACS.

https://xkcd.com/378/

Exercises: Bash

Exercises (Esercitazione0)

- Exercise 1: (`mkdirs.sh`)
  - Write a script that creates five directories named calculation_?, where ? is a number.
- Exercise 2: (`parent_script.sh`, `child_script.sh`) → Nested script
  - Write a `parent_script.sh` that executes the `child_script.sh`
  - Write a `child_script.sh` that prints out numbers from 0 to 9
- Exercise 3 : (`hello_world.sh`, `hello_world_redirect_1.sh`, `hello_world_redirect_2.sh`)
  - Create a "Hello world"-like script. Copy and alter your script to redirect output to a file using >.
  - Alter your script to use >> instead of >. What effect does this have on its behavior?
- Exercise 4 : (`generaz_num.sh`)
  - Use `seq 1 75 > numbers.txt` to generate a file containing a list of numbers. Use the `less` and `more` commands to look at it, then use `grep` to search it for a number.

  Use a `wc` to get an exact the number of lines in the file