

Introduction to Lab Sessions

PRO1

Josep Carmona, Lluís Padró

Introduction

Introduction

- In this course we will learn to write programs that run in *command-line mode* (i.e. with no GUI)
- Example:

```
int main() {  
    cout << "What is your name? ";  
    string name;  
    cin >> name;  
    cout << "Hello " << name;  
    cout << ", nice to meet you.";  
    cout << endl;  
}
```

```
$ ./hello  
What is your name? Maria  
Hello Maria, nice to meet you.  
$
```

Introduction

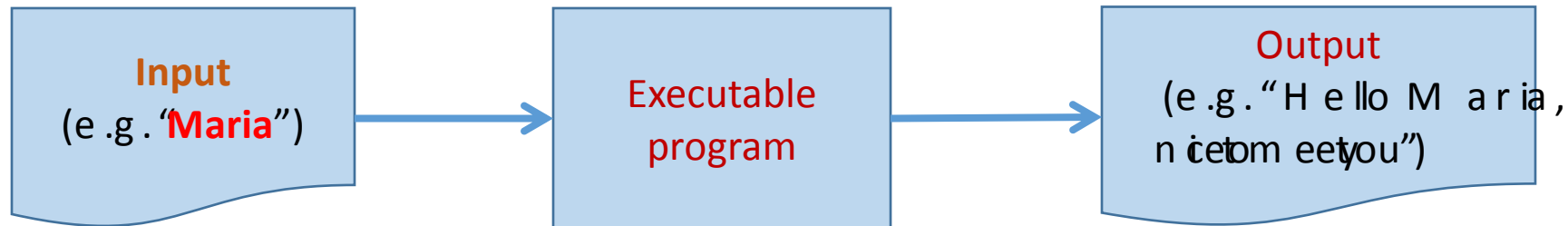
- Our programs will normally read some input (e.g. the user name in previous slide) and produce some output (e.g. the greeting)
- Example:

```
int main() {  
    cout << "Enter a number: ";  
    int n;  
    cin >> n;  
    cout << n << "x1 = " << n*1 << endl;  
    cout << n << "x2 = " << n*2 << endl;  
    cout << n << "x3 = " << n*3 << endl;  
    cout << n << "x4 = " << n*4 << endl;  
    cout << n << "x5 = " << n*5 << endl;  
}
```

```
$ ./multiply  
Enter a number: 121  
121x1 = 121  
121x2 = 242  
121x3 = 363  
121x4 = 484  
121x5 = 605  
$
```

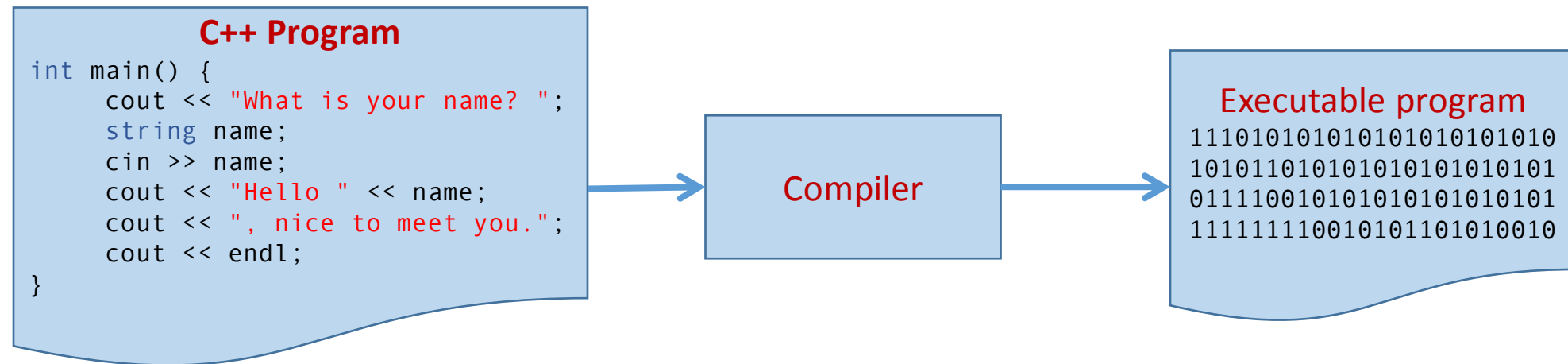
Building programs

- To be run in a computer, programs need to be in *executable* (a.k.a. *binary*) form.
- The program will read the input, process it, and produce the appropriate output

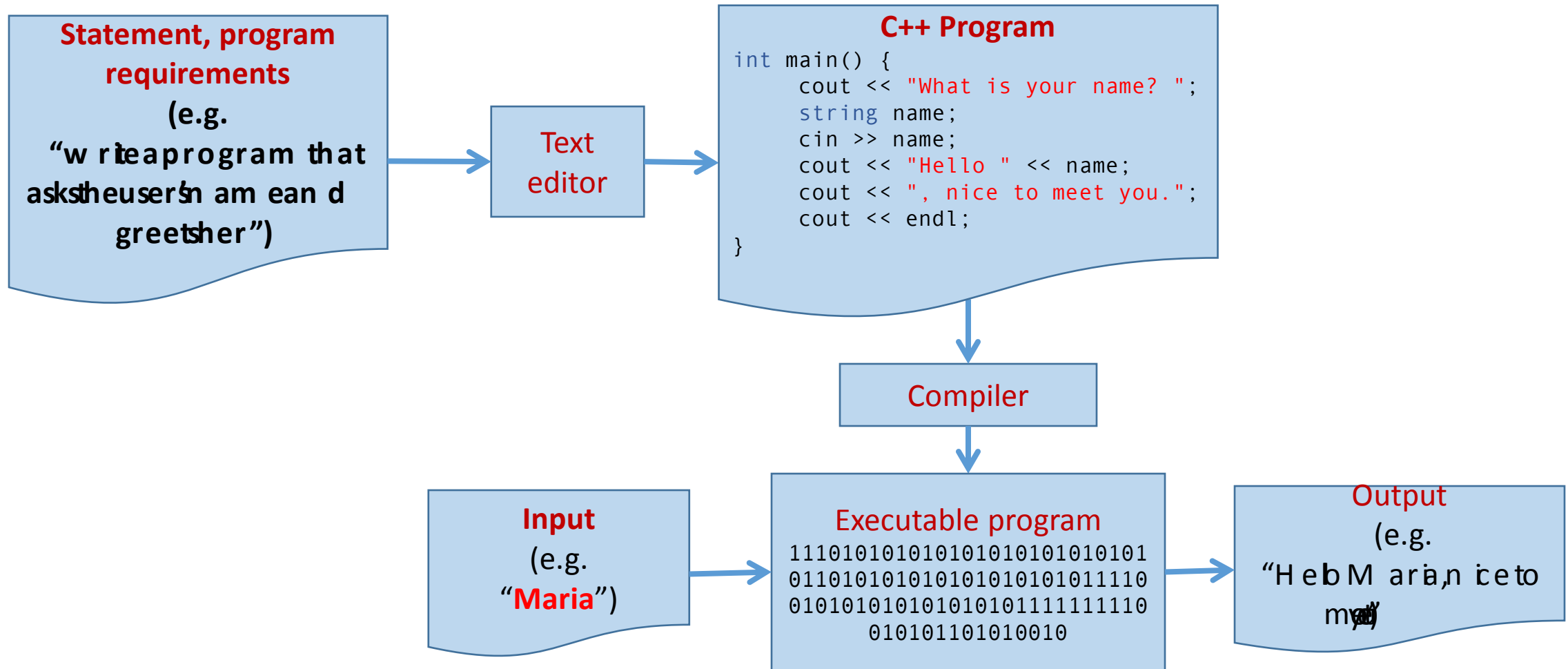


Building programs

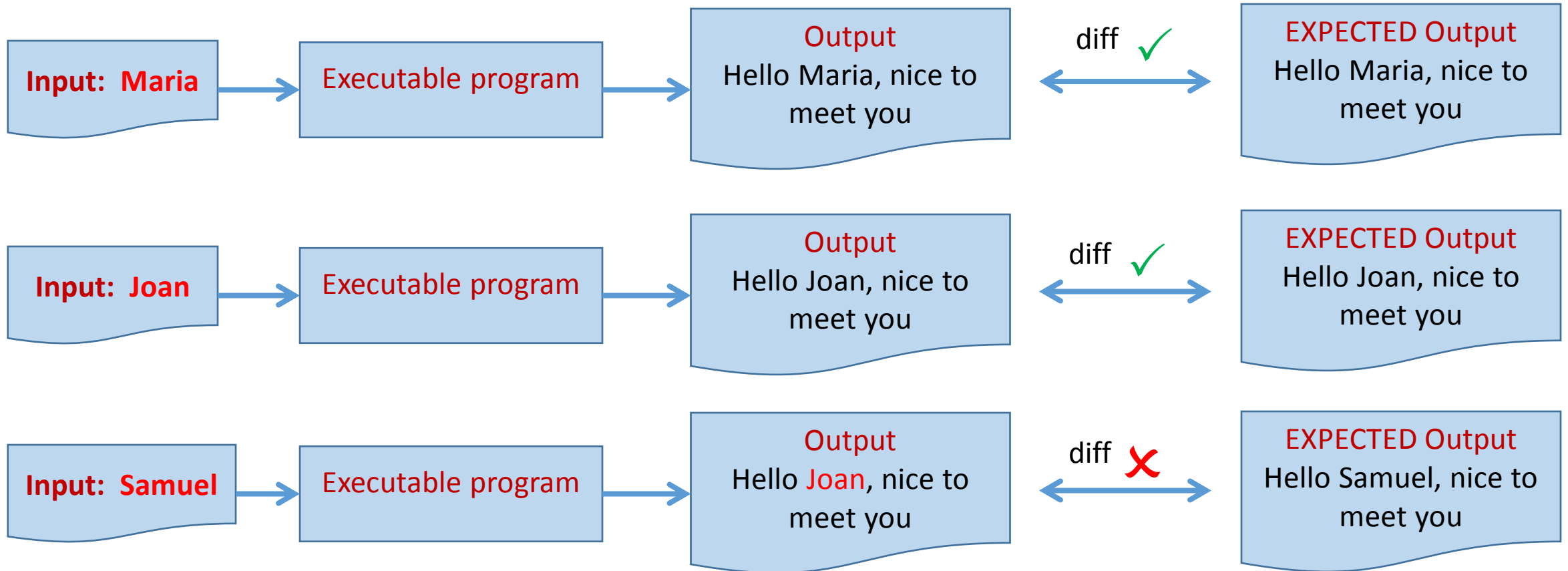
- We can not write programs directly in binary (too costly and error prone)
- Thus, we have *Programming Languages* (e.g. C++) that are closer to humans. Programs written in such languages must be converted to binary using a *compiler*.



Putting it all together



Checking that the program works



Linux

Linux desktop and command line

- In **Linux**, you have a desktop similar to that of any other O.S.
- Most tasks (**copying** or **renaming** a file, **moving** it to a different folder, **create** a new folder, etc) can be performed using the **graphical** desktop interface
- However, we are going to write **command-line** interface programs, which need to be run in a command line interpreter (also known as **console**, **terminal**, or **shell**)
- From the console, you can run commands to **execute** any program, or to **handle** files (copy, rename, move, etc).

Basic shell commands

A terminal has, in a given moment, one and only one *current working directory* (i.e. the *folder* we have currently open).

Shell commands are always referred to the current working directory

`cd dirname`

Open folder with given name

`cd ..`

Close current folder and go back to parent.

`pwd`

Print current working directory

`ls`

List contents of current directory

`mkdir dirname`

Create new directory with given name

`rmdir dirname`

Remove directory with given name

Basic shell commands (cont.)

<code>cp file1 file2</code>	Copy file1 to file2
<code>mv file1 file2</code>	Rename file1 to file2
<code>rm file</code>	Remove file
<code>more file</code>	Show content of file

Extensive and detailed step-by-step tutorial on shell commands for newbies:

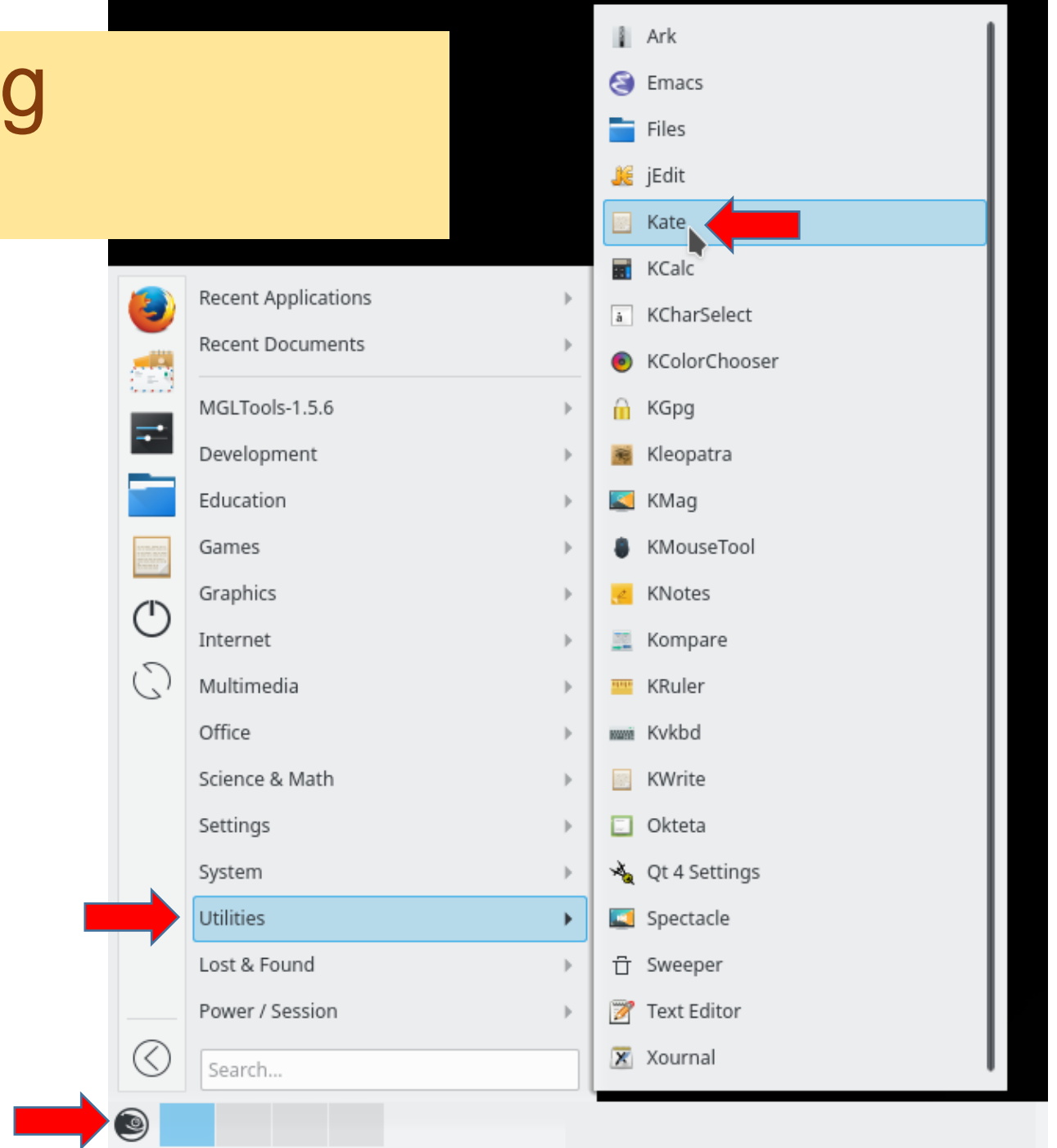
<http://linuxcommand.org/>

Writing programs

Set up programming environment

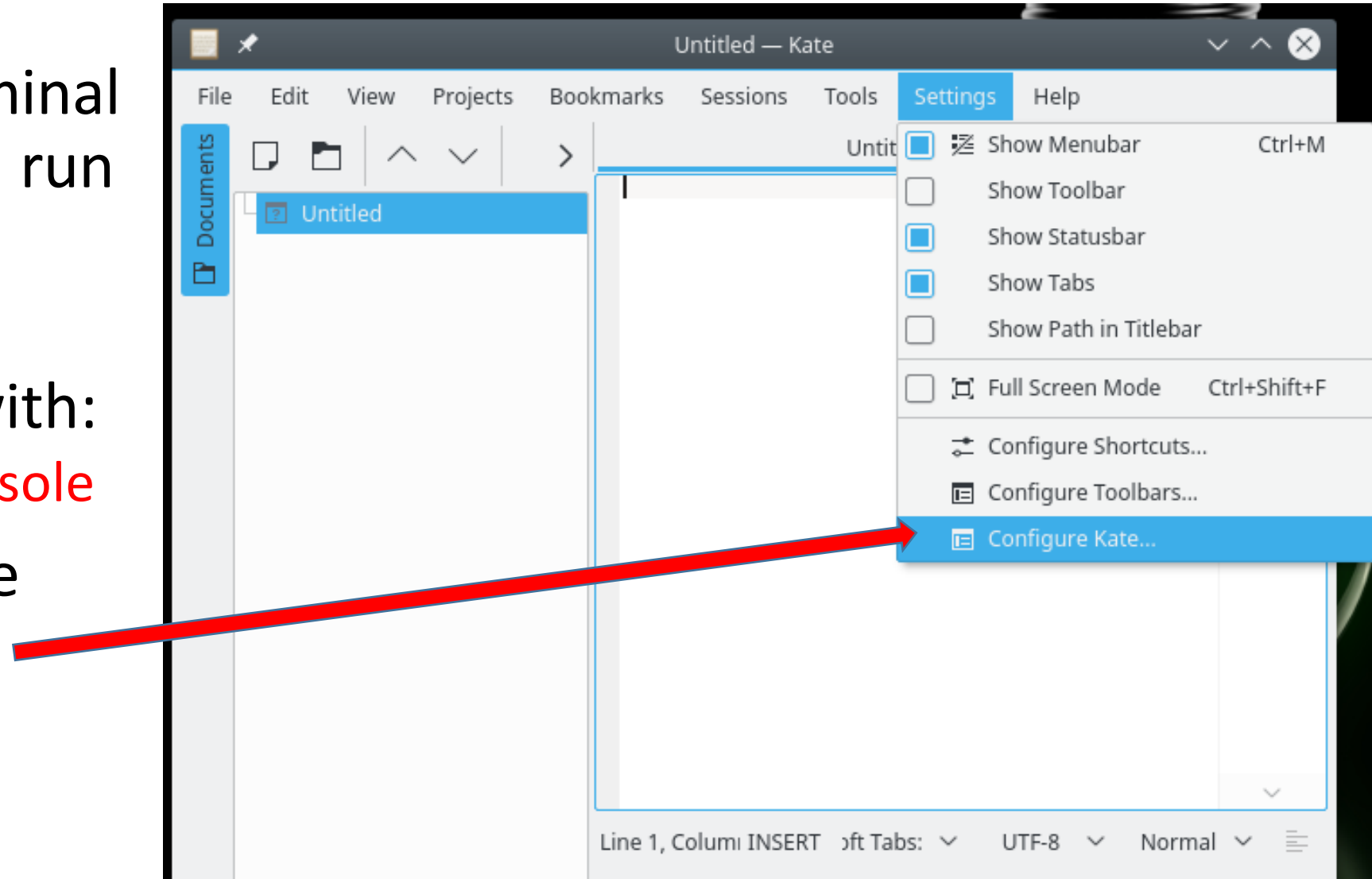
- Programs must be written on a plain text editor.
- Linux offers several of them (emacs, kwrite, TextEditor, ...).
- We recommend *kate*. Launch it from

Menu → Utilities → kate

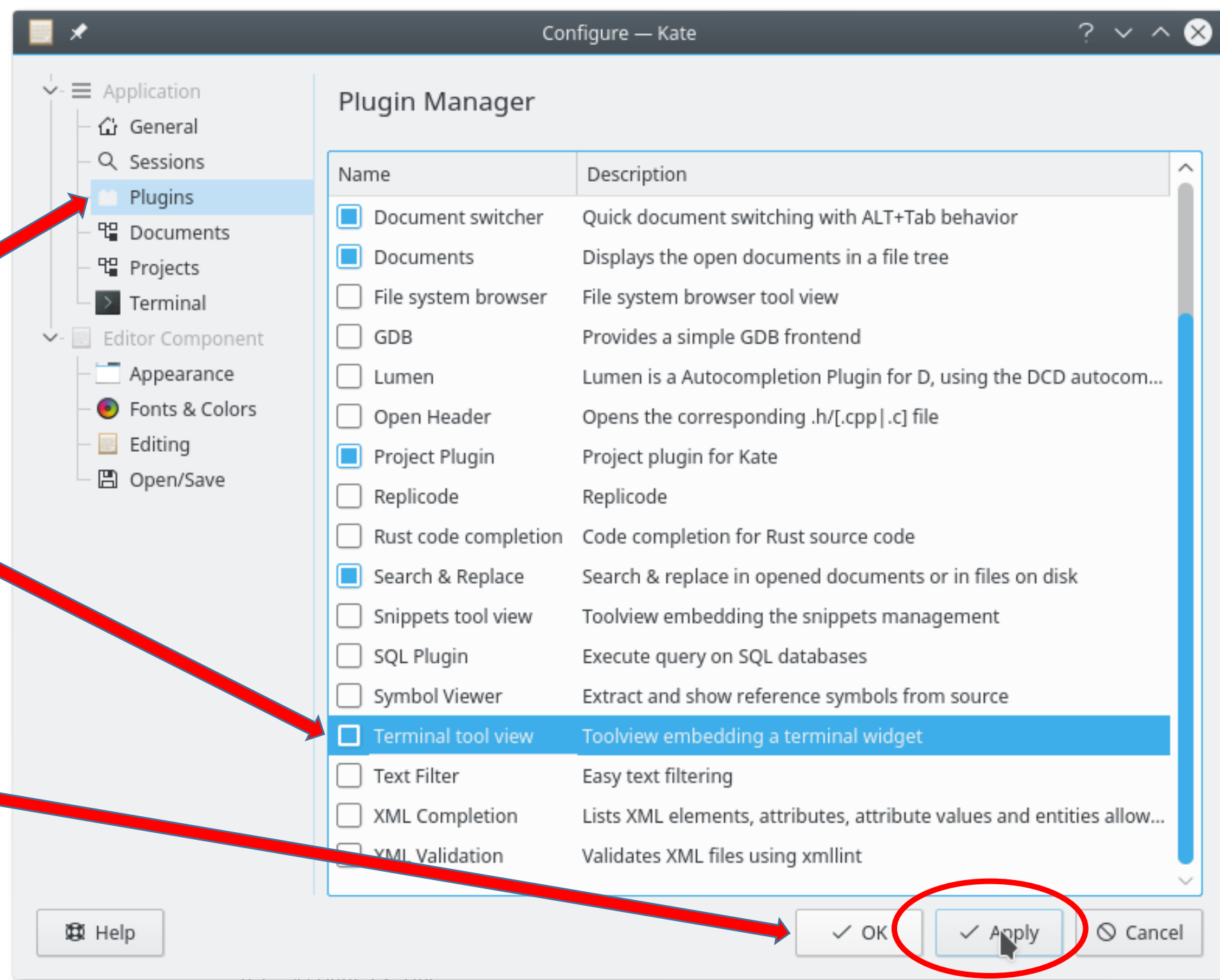


Set up programming environment

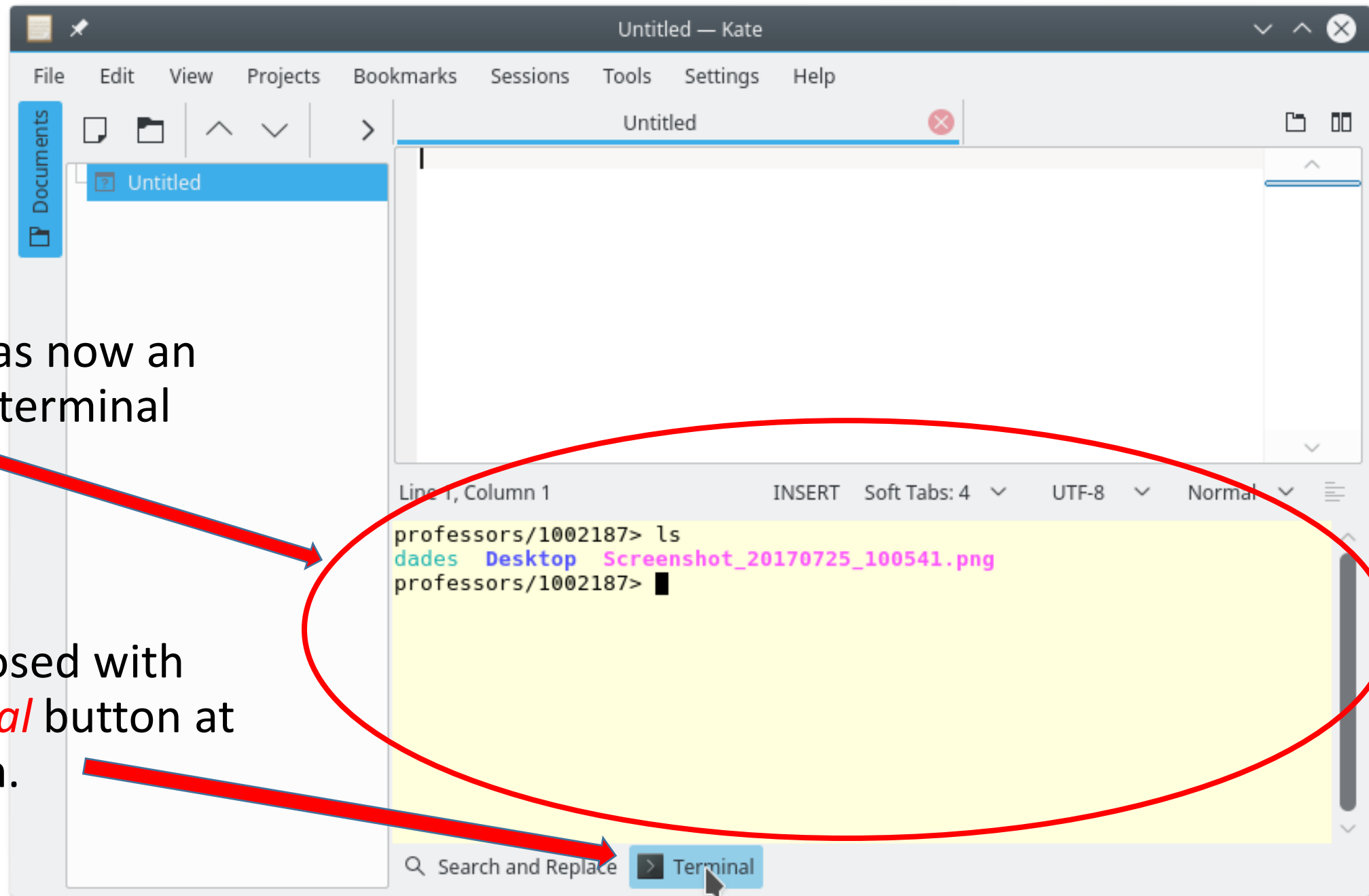
- We will need a terminal too, to compile and run our programs
- We can run it in a separate window with:
Menu→System→Konsole
- Or we can configure *kate* to show an integrated terminal



- In *kate* configuration, select *Plugins*
- Mark *terminal tool view*, near the end of the list.
- Click *Apply* and then *OK*.



- Our *kate* has now an integrated terminal window
- It can be opened/closed with the *Terminal* button at the bottom.



How to write a program

- Launch kate
- Create a new document
- write a sample program:

```
#include <iostream>
using namespace std;

int main() {
    cout << "Hello everybody!" << endl;
}
```

Save the program with a name that ends in **.cc** (e.g. hello.cc) and notice how *kate* syntax-colored the program.

How to compile a program

- Navigate in the terminal to the directory where you saved the file `hello.cc`

- Compile the program:

```
p1++ -o hello.x hello.cc
```

- If there are errors, fix them and compile again.
- Execute the program

```
./hello.x
```

Example: squares.cc

```
#include <iostream>
using namespace std;

int main() {
    int a,b,c;
    cin >> a >> b >> c;
    cout << a*a << " " << b*b << " " << c*c << endl;
}
```

```
$ g++ -o squares.x squares.cc
$ ./squares.x
6 3 12
36 9 144
$
```

Example: nif.cc

```
#include <iostream>
using namespace std;

int main() {
    int dni;
    cin >> dni;
    const string data("TRWAGMYFPDXBNJZSQVHLCKE")
    cout << "NIF letter: " << data[dni%23] << endl;
}
```

```
$ g++ -o nif.x nif.cc
$ ./nif.x
45678901
NIF letter: G
$
```

Handling compilation errors

- If there are errors, the executable is **not created**.
We must fix the errors and compile again.

```
#include <iostream>
using namespace std;

int main() {
    int a,b;
    cin >> a >> b >> c
    cout << a*a << " " << b*b << " " << c*c << endl;
}
```

```
$ g++ -o squares.x squares.cc
squares.cc:6:30: error: 'c' was not declared
squares.cc:7:3: error: expected
```

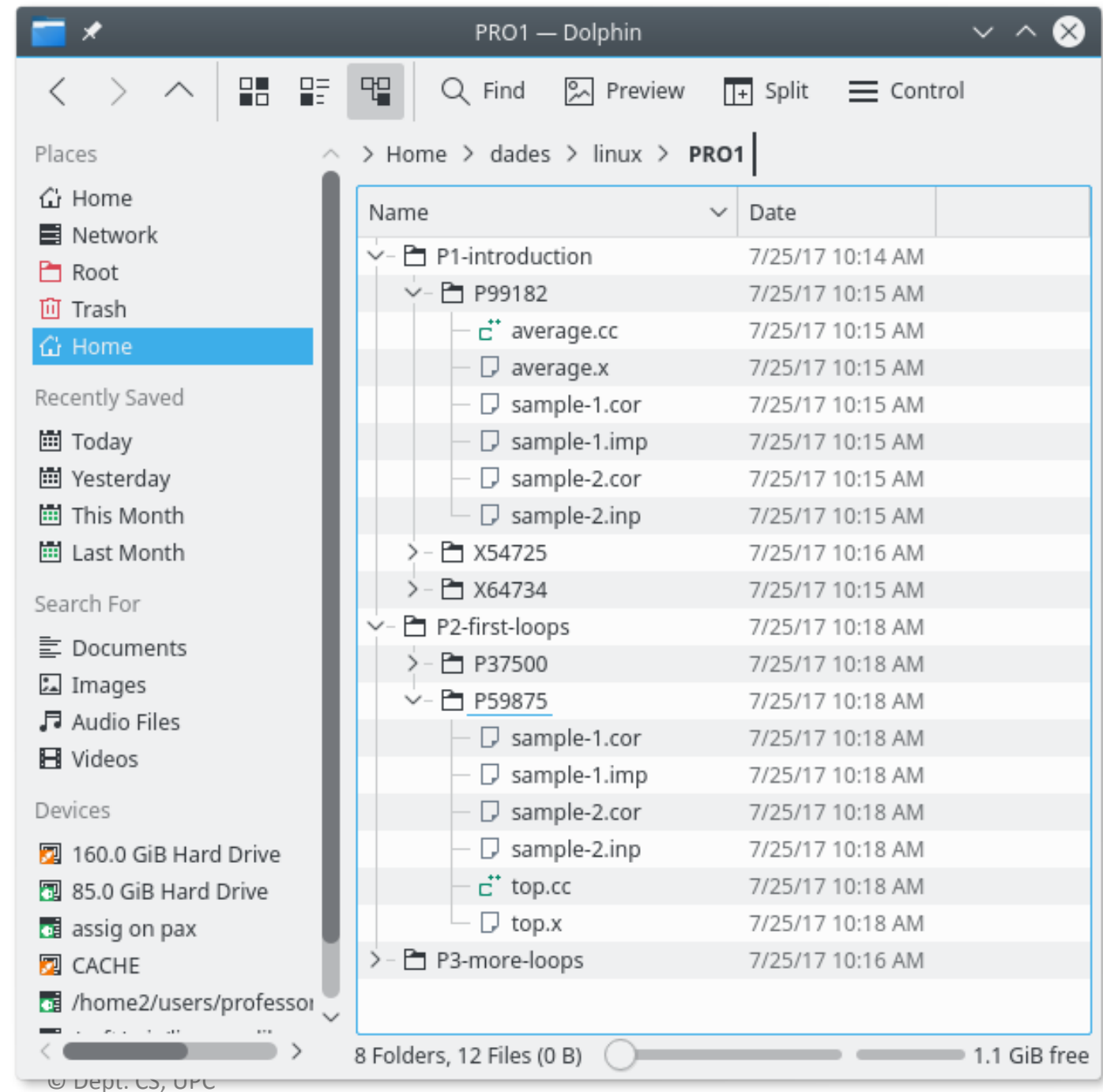
Organize your work

Organize your exercises

- During the course there will be three lab **exams**.
- There are **dozens** of exercises in the course.
- Exercises are organized in **lists**, by course chapters.
- To be allowed to take each exam, a **minimum** number of exercises must be solved in each list.
- It is crucial to have exercises **organized** to avoid getting lost.

Organize your exercises

- We recommend having a folder for each problem.
- It is also useful to group problem folders depending on the list they belong to.
- In each problem folder, you can have the C++ **program**, the **executable**, and its **input** and **output** files.



The *Jutge*

Automatic scoring of programs

- <http://jutge.org> is the environment where we will grade the **lab exercises** and we will take the **course exams**.
- You have been invited to this course. Find it in the list, and click “*enroll this course*”.
- You can **submit** your programs to the *jutge* and find out whether they work.
- You can also **download** the input files and expected outputs for each problem, to check them in your PC.
- It is important to be able to work locally: In the exams, **penalizations** are applied after three requests to the *jutge*.

Important things to know

- At <http://www.cs.upc.edu/~pro1> you will find important information about this course.
- In particular, check the tab
- “*Entregues problemes*”, which is updated frequently and contains:
 - The lists of problems in the *judge* you must solve to be admitted to each **exam**.
 - The range of **dates** when each list must be solved.
 - Which problems of each list the you have **solved** in the **required** dates.
- Check this page often! **No claims** will be accepted about problems submitted out of the required dates.

Example

- Now your professor will do an example problem on the *jutge*. Try to follow it in your computer.

Checking program results

Input/output in C++

- Read data

```
int a,b,c;  
cin >> a >> b >> c;
```

- Write data

```
int a;  
cout << "Value: " << a << endl;
```

- The output must be **exactly** as the expected for the problem to be accepted by the *judge*.

Problems with manual input/output

- Manual input

- We can not change the input once we press `return`.
- Time-consuming and error-prone when the input is long.
- We must press `ctrl-D` to end the input.

- Manual check of the output

- If the output is long, it is difficult to spot small differences with respect to expected output.

“Automatic” input/output

- Run program redirecting input and output

```
./squares.x <sample-1.inp >sample-1.out
```

Symbol `<` will read input from given file instead of keyboard.

Symbol `>` will write output to given file instead of display.

- Compare obtained output with expected output

```
kompare sample-1.out sample-1.cor
```