ESIPAP European School of Instrumentation in Particle and Astroparticle Physics

2016 edition

Supervisers: Eric CHABERT, Eric CONTE

Computing session 1

Processing data with a C++ program

Abstract:
Through this computing session, the ESIPAP students will apply their programming knowledge for instrumental purpose. The simple example in this session consists in reading data acquired by a sensor and in processing them before dumping the results in a text file. The students will apply the first notions of programming and will learn how to build a C/C++ project. The second part of the session addresses the migration of the project to modular programming. The students will learn the benefit of the *makefile*-based compilation.

Pedagogical goals:

C/C++ language

- Implementing, in an efficient way, elementary algorithms.
- Reading and writing text files.
- Manipulating data arrays.
- Handling strings and string-float conversions.
- Writing functions and making the code modular.
- Improving the robustness of the code in order to prevent abnormal termination or unexpected actions.

Compiling/linking

- Creating an executable file from a simple source file.
- Compiling and linking a project made up of several source files: in a manual or automated (Makefile) way.

Requirements:

- First notions of C/C++ programming: I/O access, arrays, pointers/references, functions.

Contents

Part I

Introduction to the ESIPAP computing sessions

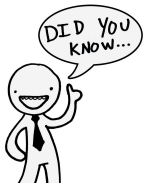
1 Foreword

Computing sessions belong to the educational program of the ESIPAP (European School in Instrumentation for Particle and Astroparticle Physics). Their goal is to teach the secrets of C++ programming through practical work in the context of high energy physics. The session is designed to be pedagogical. It is advised to read this document section-by-section. Indeed, except the *Physics context*, each section of the document is a milestone allowing to acquire computing skills and to validate them. The sections related to C++ programming are ranked in terms of complexity. In order to facilitate the reading of this document and to measure his progress, the student must **fill up the dedicated roadmap** which includes a check-list and empty fields for personal report.

In the document, some graphical tags are used for highlighting some particular points. The list of tags and their description are given below.

The student is invited to perform a pratical work by **writing a piece of code** following some instructions.

Analyzing or interpreting task is requested and the results must be reported in the roadmap.



Some **additional information** is provided for extending the main explanations. It is devoted to curious students.

A piece of **advice** is given to help the student in his task.

2 The ESIPAP framework

The practical works must be performed on devoted machines where all required software are properly installed. The user will find below all the instructions for setting the environment at each beginning of session.

2.1 Launching the `Windows` machine

You must choose a computer in the computing room, spot its name and check that no peripheral is missing (mouse, keyboard, ...). Then boot it and login to the `Windows` operator system (supervisors will provide the password access).

2.2 Accessing the `Linux` virtual machine

The practical sessions will be achieved on a `Linux` machine for pedagogical motivations. You must connect a virtual machine. First click on the "Start" button, i.e. the button with the `Windows` logo, located on the bottom left of the screen (see Figure 1).

Figure 1: The `Windows` Start button

According to Figure 2, click on the virtual machine called **"ESIPAP_slc6"**. A password could be necessary and should be supplied by the supervisors.

Figure 2: The screen showing the available virtual machines

2.3 Setting the environment

To load the work environment, you can issue the command below at the shell prompt.

```
bash$source␣/home/esipap/tools/setup.sh
```

If the system is properly installed, the version of each tool to study should be displayed at the screen like below. If you have an error, please call the supervisors.

```
-----------------------------------------------
              ESIPAP environment
-----------------------------------------------
 - GNU g++   version 4.9.1
 - ROOT      version 6.06/00
 - Geant4    version 10.2.0
-----------------------------------------------
```

You must work in your local folder. Of course, it is advised to create one folder for each practical session like: `session1`, `session2`, `session3` and `session4`. Do not overwrite or remove files that you wrote in a previous session.

2.4 Saving your work on a share disk

Your work will be evaluated from the the piece of code that you wrote. At the end of each session you must save your production on a share disk. The virtual machine is equipped with one share disk called "ESIPAP-SHARE" and saved everyday. For accessing this disk, click on the `Linux` tab named **"places"** according to Figure 3 and select the disk **"ESIPAP-SHARE"**.

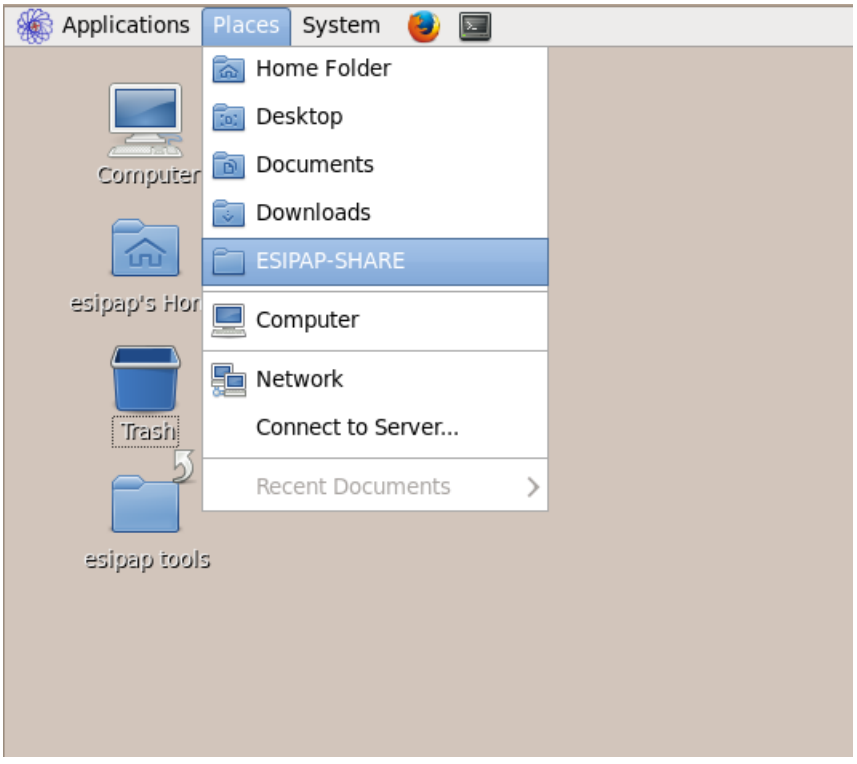


Figure 3: The ***Linux*** *tab named "places"*

After entering a password, the list of all connected machines in the room is displayed (see Figure 4). Select the folder corresponding to your machine and put there all you work. Please organize this folder by creating one folder for each practical session like: `session1`, `session2`, `session3` and `session4`.

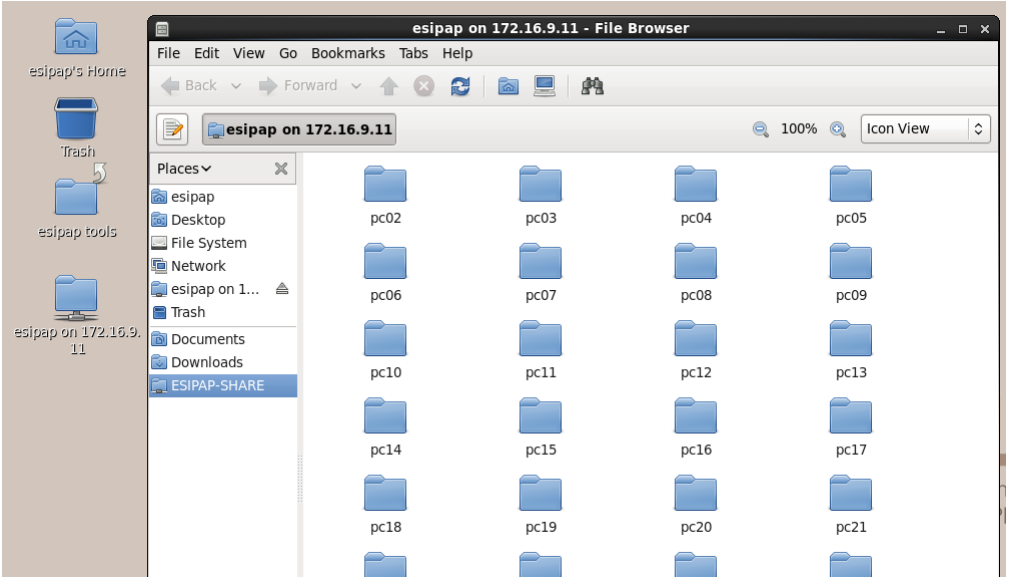


Figure 4: List of all available machines in the room

Part II
Getting started with GNU g++

3 Hello world!

In this section, a very simple example of main program (the so-called *hello world* example) is supplied and explained in order to help beginners in C++ programming. More experimented students should be a little patient: challenging tasks are coming soon.

3.1 Main program source

The main program will be contained in a source file called `main.cpp`. It displays at the screen the message *"Hello World!"*.

```
// STL headers
#include <iostream>
using namespace std;

// Main program
int main(int argc, char** argv)
{
  // Display messages at screen
  cout << "Hello World!" << endl;

  // Normal program termination
  return 0;
}
```

Listing 1: A first main program

3.2 Building the main program

To build with `g++` compiler an executable file from `main.cpp` file, the simpliest command to type at the shell prompt is:

```
bash$g++ main.cpp
```

If the `main.cpp` file compiles properly, an executable file with the default name `a.out` is created. Of course, we invite the students to use `g++` in more advanced way by adding three items:

- giving a proper name to the executable program
- splitting the complation step from the linking step
- specifying some compilation options

To avoid retyping several times during the session the building commands, a shell script can be written. This is an example of a such file called `mymake`:

```
gcc -W -Wall -ansi -pedantic -o main.o -c main.cpp
gcc -o main main.o
```

Listing 2: A first building script

It is necessary to make this script executable before launching it.

```
bash$chmod +x mymake
```

3.3 Work to do

- **Recopy the content of the `main.cpp` and `mymake` files.**
- **Build the program and test that the *"hellow world"* message appears properly when you launch the executable.**

- **Explain the compilation options used for generating the object file `main.o`.**

4 Elementary statistics on a data file

4.1 Physics context

We assume a digital sensor measuring energy deposits at a given frequency. The integer output values must be positive and be smaller than 1023. These values are saved in text files. An example of a such file containing 1000 events could be found here: `events.dat`. The file format is very simple: one datum a line.

4.2 First program

You must write a program achieving an elementary statistics study on the data file. The program will be made up of only one source file called `main.cpp`.

- **Read the data file and save all the data into an array.**
- **Compute the following statistical information:**
  - the maximum value
  - the minimum value
  - the mean value
  - the root-mean-square value
  - the median
- **Displaying at the screen these information.**

4.3 An improved code

We would like to improve the robustness of the code. For testing the improvements, the student can use the following input file: `events2.dat`.

- **The data file name should be not hard-coded. Adapt your code in order to give the file name as an argument of your program.**
- **Your text-file reader should check that each value is consistent with the range [0;1023] specified in the physics context. It must now warn the user when an irrelevant value is found and must skip it.**

5 Processing a data file

5.1 Physics context

We keep the context of the previous section. But now we must correct the output values by applying :

- an offset: a real value to subtract to the raw energy.
- a gain: a multiplicative value (defined as strictly positive real).

Thus the corrected energy (`energy*`) is the result of the following equation:

$$\text{energy}^* \ = \ (\text{energy-offset}) \times \text{gain}$$

5.2 Tasks to achieve

- **Writing a code which asks to the user to specify the values of the offset and the gain.**
- **Computing for each input energy value the corrected value.**
- **Calculating the statistical information on the corrected values and displaying them at the screen.**
- **Saving the results in an output file with the following conventions:**
  - the file name: <input name>+"_"+<average value>. The '.' character of the average value will be replace by the 'p' character. For instance: "3.14" replaced by "3p14".
  - The file should contain one line a datum in scientific notation.

Part III
Modular programming

6 Creating a project with several source files

6.1 Modular programming concepts

We would like to split the file `main.cpp` into several source files. The code should be reorganized in terms of functions. We propose the following scheme for the file naming:

- the header file `statistics.h` and the source file `statistic.cpp` containing functions related to the statistics
- the header file `processing.h` and the source file `processing.cpp` containing functions related to the energy correction algorithm.
- the header file `datafile.h` and the source file `datafile.cpp` containing functions for reading or writing a data file.
- the main file `main.cpp` calling the different functions.

6.2 Work to do

- **Change your code by making functions and put them in the different files.**
- **Adapt the script `mymake` for compiling this project with several source files.**
- **Test if the program is working properly.**

6.3 Include guard

Usually in a C++ project, the core code of a header file is contained between preprocessor directives such as the file below.

```
#ifndef MYFILE_H
#define MYFILE_H

// ......
// core code
// ......

#endif
```

Listing 3: A template of a C/C++ header file

- **What do these lines? Why are they written?**
- **Apply this structure to all your header files.**

7 Compiling with GNU MAKE

In spite of its simplicity, the shell script `mymake` used for building the executable program has two disavantages. First, each source file is compiled when the script is launched. For big project, compiling all files could take a lot of time and this time could be an issue if only one source file has been changed since the last compilation. Secondly, the compilation command must be repeated in the script as many time as there are source files. Besides, new compilation commands must be added if new source files are created. The manual writing and management of this script should be painful in the context of big project.

To tackle these two disavantages, project building can be performed by using an advanced configuration file containing generic and compact compilation instructions. This kind of configuration file is called `makefile`. Numerous programs allow to interpret the makefile and to launch automatically the compilation sequence: `GNU make` (called also `gmake`), `nmake`, `tmake` ... and, unfortunately, each corresponding makefile has a specific syntax. The following explanations are based on the example of the most popular tool: `GNU make`.

7.1 Some words about the program `GNU make`

The `GNU make` tool is usually included in every LINUX distributions and it is fully operational on LXPLUS session of the students. The corresponding executable program is called `gmake` or simply MAKE. To check the presence of this program, you can issue the command below at the shell prompt: the release version must be displayed at the screen.

```
bash$make -v
```

By default, `GNU make` will look for a makefile called `Makefile` or `makefile`. The next sections of this document are devoted to the syntax of this file.

7.2 Minimal makefile

Here is explained the simplest way to write a makefile. For explaining the syntax, let has consider the example of a project made up of a main source file called `main.cpp` which use two classes described in the header/source files `file1.h`, `file1.cpp`, `file2.h` and `file2.cpp`. Building an executable program called `main` can be performed with the following makefile:

```
1  # Makefile example
2
3  all: main
4
5  main.o: main.cpp
6  	gcc -W -Wall -ansi -pedantic -o main.o -c main.cpp
7
8  file1.o: file1.cpp file1.h
9  	gcc -W -Wall -ansi -pedantic -o file1.o -c file1.cpp
10
11 file2.o: file2.cpp file2.h
12 	gcc -W -Wall -ansi -pedantic -o file2.o -c file2.cpp
13
14 main: main.o file1.o file2.o
```

```
15 ␣␣␣␣gcc␣-o␣main␣main.o␣file1.o␣file2.o
16
17 clean:
18 ␣␣␣␣rm␣-rf␣*.o␣main
```

Listing 4: A simple makefile

Like for shell script, lines begun with # are interpreted as comment lines. The file is made up of several instruction blocks called *rules*. Each *rule* targets to compile a source file or to link the object files. The generic syntax for a *rule* is the following:

```
1 target:␣dependency1␣dependency2␣[...]
2 ␣␣␣␣instructions1
3 ␣␣␣␣instructions2
4 ␣␣␣␣[...]
```

When `GNU make` treats a *target*, it analyzes first the *dependencies*. If a *dependency* is a file, the program determines if this file has been changed since the previous compilation. If a *dependency* is a target specified in the makefile, the program checks if the target has been treated. In the case of one dependency has changed or has to be rebuilt, `GNU make` treats the *target* before and execute the *instructions*. In the other case, the instructions are skipped. **Beware: instructions are preceded by a tabulation character (and not by space characters).**

To launch `GNU make` and to interpret the makefile, just type the following command at the shell prompt:

```
bash$make
```

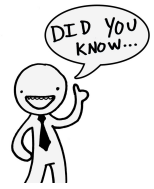
`GNU make` looks for the makefile and treats the first rule specified in the makefile. Usually it is called *all*. Of course, a given target could be specified to the program by set the target name as an argument of `make` command. This is the example of application to the target `main`:

```
bash$make␣main
```

We focus the user that the following makefile contains a specific rules called `clean`. It is very useful to remove files produced by *g++* (object files *.o and executable program) in order to back to the original source.

```
bash$make␣clean
```

- **By analyzing the example above, write a makefile adapted to the programming project.**
- **Clean your project with the makefile rule `clean` (a copy of the source file must be saved in a safe place in case of an unexpected deletion due to a bug in the makefile).**
- **Compile the project with the makefile.**
- **Check that if only one file is changed, only source depending on this file are treated in the next `GNU make` run.**



Some developers prefer splitting the `clean` target into two targets: `clean` for removing only temporary files (object files) and `mrproper` for removing all compiler produced files (object files and executable).

7.3 Enriched makefile

The previous makefile example is not really automated. In the next example, internal variables are used and allow to write compact and generic rules.

```
1  # Makefile example using variable
2
3  CC=g++
4
5  CFLAGS  = -W -Wall -ansi -pedantic
6  SRCS = $(wildcard *.cpp)
7  HDRS = $(wildcard *.h)
8  OBJS = $(SRCS:.cpp=.o)
9  EXEC=main
10
11
12 all: $(SRCS) $(EXEC)
13
14 $(EXEC): $(OBJS)
15 	$(CC) $(LDFLAGS) $(OBJS) -o $@
16
17 %.o: %.cpp %.h
18 	$(CC) $(CFLAGS) -c $< -o $@
19
20 clean:
21 	rm -f *.o $(EXEC)
```

Listing 5: an automated makefile

Definition of variables follows the scheme *VARIABLE = value*. The list of variables used in the analysed makefile can be found below. Of course, the user can define his/her own variables.

- *CC*: compiler command
- *CFLAGS*: compiler options
- *SRCS*: list of source files (*.cpp).
- *HDRS*: list of header files (*.h).
- *OBJS*: list of object files (*.o).
- *EXEC*: name of the executable program to create.

To access the content of a variable, the syntax is: `$(VARIABLE)`. For information, the special value `$(wildcard *)` is very useful because it allows to extract a list of files from the local

folder safisfying a given criterion.

Then there are also some special variables, internal to `GNU make` which can be used in the different *rules*. The two such variables used in the example are very powerful:

- *$@*: name of the *target*.
- *$<*: name of the first dependency.

Finally, repeating rule definition could be avoided by using automated rules. Thus, the following rule is applied to every file ended with '.o'. The character % replace the name of the files.

```
1 %.o: %.cpp %.h
2 	commands1
3 	commands2
4 	[...]
```

- **Adapt (if necessary) the automated makefile to the programming project.**
- **Compile your program with the obtained makefile**