ESIPAP European School of Instrumentation in Particle and Astroparticle Physics

2014 edition

Supervisers: Eric CHABERT, Eric CONTE

Computing session 2

Advanced C++ model for both calorimeters of the CMS detector

Abstract:
This computing session is the direct sequel of Computing Session 1. Where as only the electromagnetic barrel calorimeter was modeling, the students must take into account the hadronic barrel calorimeter by extending the code previously written. The exercise will allow to pratice some advanced concepts of C++. In a second part, energy acquired by each (electromagnetic and hadronic) calorimeter cell will be used to reconstruct photon object by designing and writing a clustering algorithm.

Pedagogical goals:

C++ language	• Using inheritance in the code design in order to facilitate the creation and the maintain of classes. • Using polymorphism to supply a common interface to several classes. • Designing an algorithm and implementing implement it in an efficient and optimized way.
Collaboration work	• Respecting a given set of programming rules and conventions. • Generating automatically the reference documentation related to the code with DOXYGEN.
Compiling/linking	• Compiling and linking a project made up of several source files in an automated (Makefile) way.

Requirements:

- The class `CaloCell` from Computing Session 1.
- Concept of inheritance and polymorphism in C++.

Contents

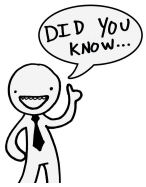
1 Foreword

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.

Concerning the evaluation of these computing sessions, all source files and other relevant digital documents must be provided to the examiner. Therefore they must be stored in a public folder on the LXPLUS session. The suggested naming convention is the following:

```
$HOME/public/TP1
$HOME/public/TP2
$HOME/public/TP3
$HOME/public/TP4
```

The student is invited to develop his code directly in such folder.

2 Physics context

The physics context is the same than the one presented in Computing session 1. Therefore it is based on the calorimetry system of the CMS detector. Where as the previous session was restricted to the electromagnetic calorimeter, the context will be extended to the hadronic calorimeter. Details on this apparatus is given in this section.

2.1 The hadronic calorimeter of the CMS detector

The aim of the hadronic calorimeter is to measure the energy of hadrons produced during the collisions. It is a sampling calorimeter, *i.e.* it is made up of alternating layers of absorber (mainly brass) and scintillator (plastic) materials. At high energies, hadrons induce hadronic shower when they interact with the absorber part of the calorimeter. The shower could be also initiated inside the electromagnetic calorimeter. The total absorber thickness at 90ř is 5.82 interaction lengths λ_i with λ_i equal to 16.42 cm. The CMS hadronic calorimeter is hermetic and compact. It covers the full range in azimuthal angle and the pseudorapidity range $|\eta| < 3$. The hadronic calorimeter is compound of multilayer *cells* and they are layout into two different geometries:

- the cylinder part, called *barrel*, has a radius of 1.77 cm and it contains 2160 cells. It covers the range $|\eta| < 1.3$.
- the two planes at each end of the cylinder (z=-4 m and z=+4 m), called *end-cap*, and contain together 1440 cells. They cover the range $1.3 < |\eta| < 3.0$.

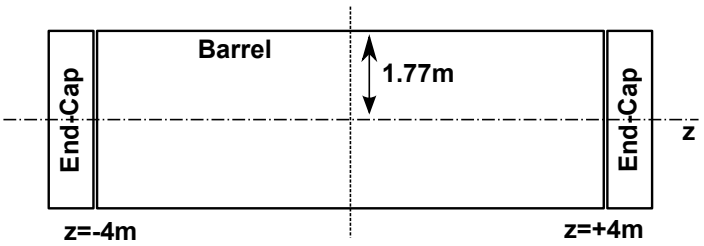


Figure 1: Barrel and End-cap part of the calorimeter in the transverse plane of the detector

Only the barrel part of the calorimeter is considered in the following.

2.2 Layout and mechanics of the barrel calorimeter

The cells are gathered in 36 wedges. One wedge contains 4×15 cells. Their layout in the $\eta - \phi$ plane is shown by the figure below.

2.3 Data acquisition by a calorimeter cell

For the sake of completness, the acquisition chain of a calorimeter cell is briefly discussed. The scintillator material emits scintillation light which is collected by WaveLength Shifting (WLS) fibres and then reaches hybride photodiodes(HPD). Reaching a Very-Front-End electronics, the signal is shaped and then digitized by an ADC (Analogic Digital Converter). After an adaptation of the signal, the signal is sent to a Front-End electronics board which computes some information useful for the first level of trigger. If the trigger is fired, digital data are sent to

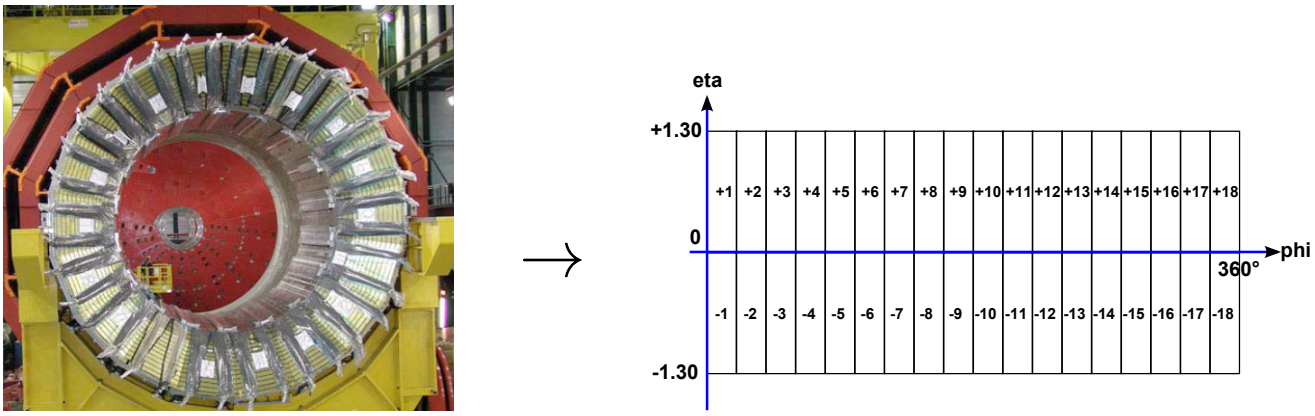


Figure 2: Segmentation of the calorimeter barrel in terms of wedges

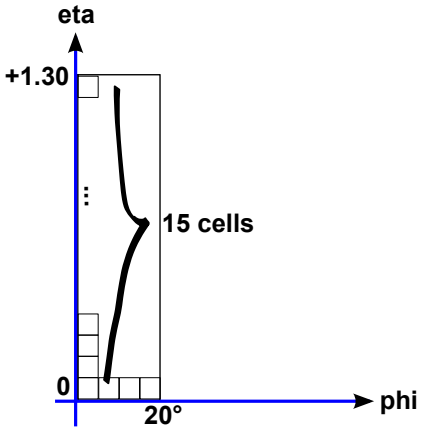


Figure 3: Segmentation of the wedges in cells

the DAQ (Data AcQuisition).

The energy resolution can be parametrized as in the equation:

$$\left(\frac{\sigma}{E}\right)^2 = \left(\frac{S}{\sqrt{E}}\right)^2 + C^2$$

where S is the stochastic term and C the constant term. Typical values are S=65% and C=6% for E in GeV.

3 Smart description of hadronic/electromagnetic calorimeter cell

The aim of this section is to implement a class describing a cell from the electromagnetic calorimeter and a cell from hadronic calorimeter.

3.1 Specifications

The class related to the electromagnetic calorimeter has been (normally) implemented in Computing Session 1 and it was called `caloCell`. In this part, in order to distinguish electromagnetic and hadronic cell, this class will be called `elecCell` in the following. The diagram UML related to this class was given in Computing Session 1 and it is reminded here:

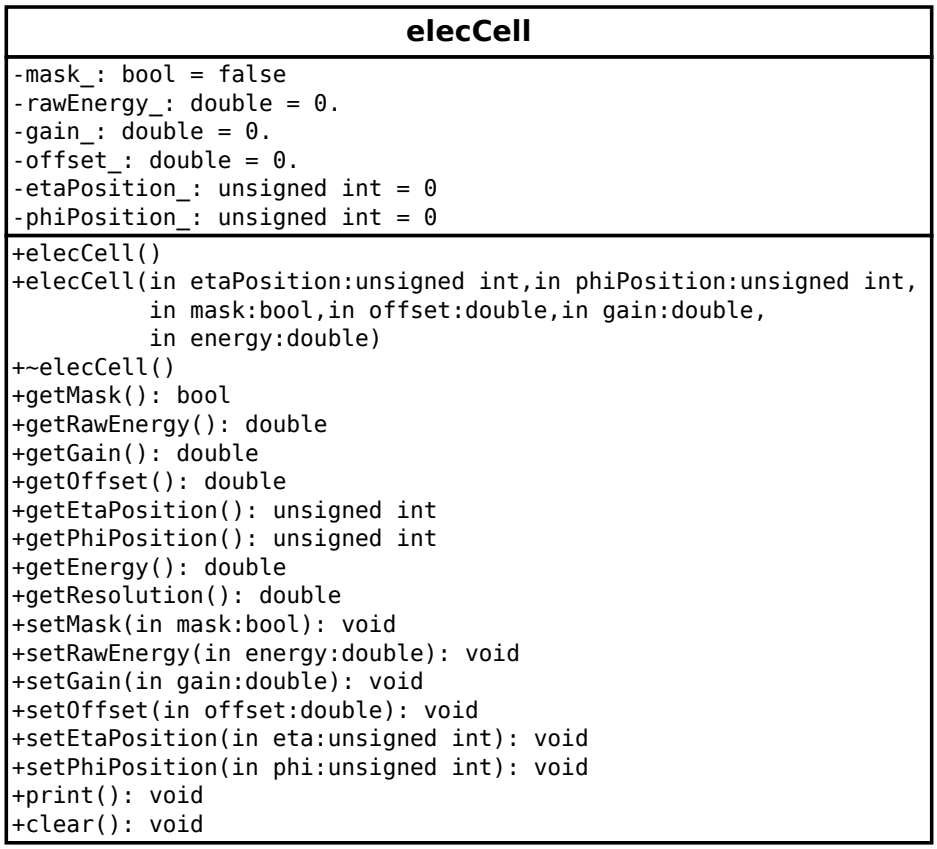


Figure 4: UML diagram related to the naive implementation of the class `elecCell`

The class describing a hadronic calorimeter cell will be called `hadCell`. The content of this class is very similar to `hadCell` but there are some differences:

- We assume the energy of the hadronic calorimeter is already corrected. So the class must not contain offset, gain and mask. The data member describing the (already corrected) energy is still called `rawEnergy_` by analogy with the class `elecClass`. Both functions `getRawEnergy` and `getEnergy` return simply the value of the data member `rawEnergy_`.
- The formula used by the `getResolution` is different than the one used for a electromagnetic calorimeter cell.

- The class has a data member specific to hadron collider: `thickness_` describing the thickness of a cell in terms of interaction length λ_i. Corresponding accessor (getter) and mutator (setter) must be implemented (functions `getThickness` and `setThickness`).

Taking into account these specifications, the corresponding UML diagram should be:

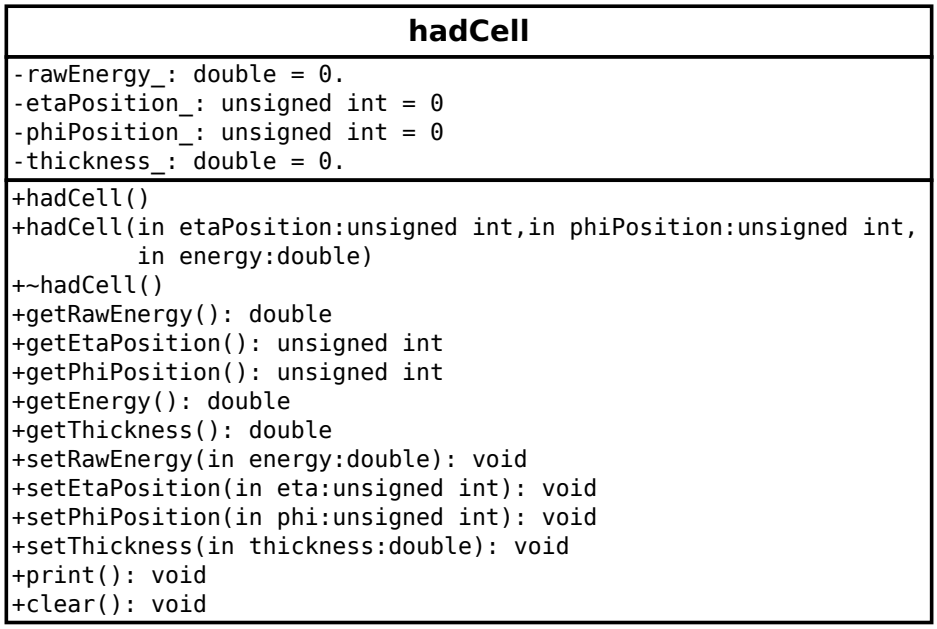


Figure 5: UML diagram related to the naive implementation of the class `hadCell`

Of course, it is possible to implement these two classes according to their UML diagrams. But we invite the students to implement them in a more efficient and clever way based on enheritance.

3.2 Class designs based on enheritance

Why the implementation described above is naive?

- The two classes have many common data member and methods. We would like to avoid as much as possible from code duplications, essentially for maintenance reasons.

- For using the polymorphism concept (explanations are given in the next section).

A more clever implementation consists in defined a third class called `caloCell`. This third class will contain all common content. The class `elecCell` and `hadCell` will inherant from the class `CaloCell`. Only their specifities will be implemented explicity. The relationship between theses 3 classes is represented in UML language by the following scheme.

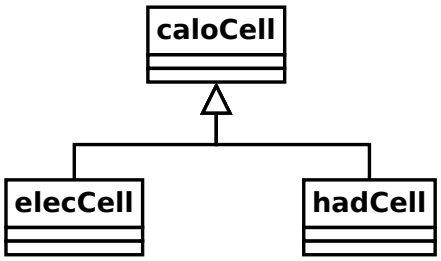


Figure 6: UML representation of the heritance relationship between the class `caloCell`, `elecCell` *and* `hadCell`

3.3 Work to do

1. **Completing the UML diagram shown by Figure 6 by adding data members and methods. This diagram UML must be equivalent to the combination of the ones shown in Figure 4 and Figure 5**
2. **Remind the 3 kinds of inheritance relationship available in C++ and their differences. What is the most appropriate one to our case?**

1. **Implement the class `caloCell` in the files `caloCell.h` and `caloCell.cpp`. It is advised to copy/paste at most from the existed code.**
2. **Check if the implementation of the class `caloCell` compiles properly.**
3. **Implement the class `elecCell` in the files `elecCell.h` and `elecCell.cpp`. It is advised to copy/paste at most from the existed code.**
4. **Test the implementation of the class `elecCell` by instantiating an object.**
5. **Implement the class `hadCell` in the files `hadCell.h` and `hadCell.cpp`. It is advised to copy/paste at most from the existed code.**
6. **Test the implementation of the class `elecCell` by instantiating an object.**

4 Array of electromagnetic and hadronic cells

We would like to apply the *polymorphism* concept through a very simple example: an array where can be stored electromagnetic cells and hadronic cells.

4.1 Preliminary

Please check that the functions `print` and `getResolution` are implemented into the 3 classes `caloCell`, `elecCell` and `hadCell`.

If it is not the case, implement them. As the function `getResolution` inside `caloCell` has no physical sense, we decide that this function will return the value -1.

4.2 First contact with polymorphism

Let's consider the following C++ code inside the `main` function:

```
elecCell* cell1 = new elecCell();
hadCell*  cell2 = new hadCell();
caloCell* theCell = cell1;
```

1. **Implement this piece of code in your `main` function.**
2. **Which methods are accessible from the pointer `theCell`? the ones from `elecCell` or the ones from `caloCell`? Test with your code.**
3. **Normally the `print` and `getResolution` functions are accessible from the pointer `theCell`. But is it the implementation of `elecCell` or `caloCell`? Test with your code**
4. **Redo the previous instructions by defining the `theCell` pointer by : `caloCell* theCell = &cell2;`**

4.3 Use virtual functions

1. **Make *virtual* the functions `print` and `getResolution` in the 3 classes.**
2. **How the `virtual` property affects the behaviour of the previous piece of code?**

4.4 Array of calorimeter cells

1. **In the `main` function, implement an array of `caloCell*` and fill this array with some example pointers to `elecCell` and pointers to `hadCell`.**
2. **Test the implemented code.**
3. **An given item of the array is a pointer `caloCell*` type. How can we determine if the pointed object is an object instantiated from the class `elecCell` or an object instantiated from the class `hadCell`**

4.5 Abstract base class

It the previous piece of code, it is possible to add in the array: pointer to `elecCell`, pointer to `hadCell` but also pointer to `caloCell`. We would like to forbid to add a pointer to `caloCell` because this class does not represent a physical cell: it is just the base class used for designing `elecCell` and `hadCell`. It can be do by making *abstract* the class `caloCell`.

1. **Explain how it is possible to make *abstract* a class.**
2. **Make *abstract* the class `caloCell` and test that all works properly**

5 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`.

5.1 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.

5.2 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 `class1.h`, `class1.cpp`, `class2.h` and `class2.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  class1.o: class1.cpp class1.h
9  	gcc -W -Wall -ansi -pedantic -o class1.o -c class1.cpp
10
11 class2.o: class2.cpp class2.h
12 	gcc -W -Wall -ansi -pedantic -o class2.o -c class2.cpp
13
14 main: main.o class1.o class2.o
```

```
15 ␣␣␣gcc␣-o␣main␣main.o␣class1.o␣class2.o
16
17 clean:
18 ␣␣␣rm␣-rf␣*.o␣main
```

Listing 1: 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
```

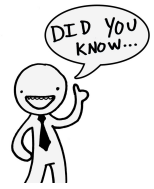
`GNU make` looks for the makefile and treats the main rule always called *all*. Of course, a given target could be specified to the program by set the target name as an argument of `make` command. This is the example of application to the target `main`:

```
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).

5.3 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 2: 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**