ESIPAP European School of Instrumentation in Particle and Astroparticle Physics

2015 edition

Supervisers: Eric CHABERT, Eric CONTE

Computing session 2

C++ model for the electromagnetic barrel calorimeter of the CMS (Compact Muon Solenoid) detector

Abstract:
This computing session is dedicated to the first notions of oriented-object programming. The physics topic chosen for the exercise is the electromagnetic calorimeter of the CMS experiment. In a first part, the students are invited to program a C++ model of the calorimeter from UML (Unified Modeling Language) diagrams. The developed code must describe the apparatus geometry, read data acquired by all cells and correct these data with calibration settings. The second part of the session consists in equipping the programming project with a *makefile*-based compilation and with an automatically generated documentation.

Pedagogical goals:

C++ language

- Writing new classes from UML diagrams.
- Instantiating objects from classes and initializing them.
- Reading and adapting an existing piece of code.
- Improving the robustness of the code in order to prevent abnormal termination or unexpected actions.

Collaboration work

- Respecting a given set of programming rules and conventions.
- Generating automatically the reference documentation related to the code with DOXYGEN.

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:

- Concept of class in C++, including constructors, destructor, mutators, accessors, ...
- Some particular C++ points: I/O access, arrays, pointers/references.

Contents

1 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`.

1.1 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.

1.2 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
```

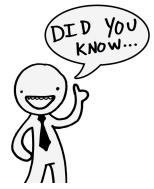
`GNU make` looks for the makefile and treats the main rule always called *all*. Of course, a given target could be specified to the program by set the target name as an argument of `make` command. This is the example of application to the target `main`:

```
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).

1.3 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**

2 Generating documentation from C++ sources

Annotation and comments inside the code is very useful for the understanding. In order to increase the documentation level, it is also possible to generate automatically reference documentation by reading the syntax and the annotations of the code. Whereas some documentation generators such as JAVADOC are specific to one programming language, the DOXYGEN program has the advantage to be used for plenty languages.

2.1 First words about the DOXYGEN package

DOXYGEN can read not only C++ language but also JAVA PYTHON, FORTRAN, PHP and others. The formats of the generated documentation are mainly HTML and LATEX (PDF or PS after LATEX compilation). It can cross reference documentation and code, so that the reader of a document can easily refer to the documentation.

The package can be downloaded from the official website (`www.doxygen.org`). From the LXPLUS session, DOXYGEN program can be launched from any folder. A small test to check the presence of this package consists in issuing the command below at the shell prompt. If the program is found, the version release must appear at the screen.

```
bash$doxygen␣--version
```

2.2 Standard doxygen configuration file

The starting point consists in writing a DOXYGEN configuration file. A template of a such file can be generated by typing the following command:

```
bash$doxygen␣-g␣doxygen.cfg
```

A text file called `doxygen.cfg` is then created and can be modified with a text editor. It contains all the available `Doxygen` options set with the default values. The syntax is very similar to a shell script. To enter into details, comment line begins with a # character and options are specified by the scheme `tag = value`. The options values are usually the reserved words `YES` or `NO` for binary options, or string for other option kinds. Appearance order of the options is not relevant.

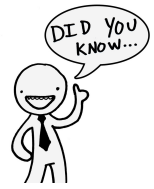
For generating HTML, the user must set the following settings:

```
1 GENERATE_HMTL␣=␣YES
2 HMTL_OUTPUT␣=␣html␣#␣name␣of␣the␣folder␣where␣HTML␣document
3 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣#␣will␣be␣generated
```

and for LATEX, the following lines

```
1 GENERATE_LATEX␣=␣YES
2 LATEX_OUTPUT␣=␣latex␣#␣name␣of␣the␣folder␣where␣LATEX␣document
3 ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣#␣will␣be␣generated
```

By default, all source files (C++ and other programming languages) placed in the local folder are taken into account. These properties can be tuned by changing options such as `FILE_PATTERNS`, `RECURSIVE` and `EXCLUDE`.



A GUI (Graphical User Interface) wizard configuration tool, called `doxywizard`, exists also. It facilitates the DOXYGEN configuration and running. Nonethess this program is not installed on LXPLUS session.

2.3 Adding graphics in the reference documentation

DOXYGEN tool can use GRAPHVIZ package for generating graphs and diagrams. It can be downloaded from the official website (`http://www.graphviz.org/`). From the LXPLUS session, GRAPHVIZ is already installed and ready to used. A small test to check the presence of this package consists in issuing at the shell prompt the command below. The version of GRAPHVIZ must appear at the screen.

```
bash$dot -v
```

For enabling all the graphical options in the report, the user must apply the following settings:

```
HAVE_DOT                = YES
CLASS_GRAPH             = YES
COLLABORATION_GRAPH     = YES
GROUP_GRAPHS            = YES
UML_LOOK                = NO
TEMPLATE_RELATIONS      = YES
INCLUDE_GRAPH           = YES
INCLUDED_BY_GRAPH       = YES
CALL_GRAPH              = YES
CALLER_GRAPH            = YES
GRAPHICAL_HIERARCHY     = YES
DIRECTORY_GRAPH         = YES
DOT_MULTI_TARGETS       = YES
```

2.4 Launching DOXYGEN

To generate automatically documentation, the user has just to type the `Doxygen` command following the name of the configuration file:

```
bash$doxygen doxygen.cfg
```

During the documentation generation, error or warning could be displayed. The user is invited to read these messages and to investigate the relevant ones. If the running is successful, folders `html` and `latex` are generated according to the configuration file.

- `html` folder contains all HTML files and can be browsed with a navigator internet from the file `index.html`.

- `latex` folder contains `latex` files and can be compiled with latex with a makefile. By issiung the command `make`, a PDF file is created and can be viewed with a PDF reader.

2.5 Work to do

- **Generate the documentation related to your code in LaTeX and HTML format**
- **Add/adjust annotations in your code in order to improve the generated documentation.**

Some suggestions about the documentation layout:

```
FULL_PATH_NAMES ======== = NO
JAVADOC_AUTOBRIEF ====== = YES
HIDE_UNDOC_CLASSES ===== = NO
GENERATE_LATEX ========== = NO
TAB_SIZE ============== = 4
OPTIMIZE_OUTPUT_FOR_C === = YES
BUILTIN_STL_SUPPORT ==== = YES
EXTRACT_ALL ============ = YES
RECURSIVE ============== = YES
SOURCE_BROWSER ========= = YES
ALPHABETICAL_INDEX ===== = YES
GENERATE_TREEVIEW ====== = YES
TEMPLATE_RELATIONS ===== = YES
SEARCHENGINE =========== = YES
REFERENCED_BY_RELATION = = YES
```