Code conventions

Nicolas Kielbasiewicz

May 15, 2014

1 Why code conventions ?

The question seems trivial, even needless, but is truly fundamental. The reasons, shown for example on the Sun homepage[1] are the followings :

- 80% of the lifetime cost of a piece of software goes to maintenance.
- Hardly any software is maintained for its whole life by the original author.
- Code conventions improve the readability of the software, allowing engineers to understand -new code more quickly and thoroughly.
- If you ship your source code as a product, you need to make sure it is as well packaged and clean as any other product you create.

The purpose of these conventions is to centralize information, to make decisions when ambiguities may exist, and so as to homogenize contributions of different programmers of a team, to make maintenance easier and to enable identification of code specific programming elements versus language specific programming elements.
The last point is the most important when we develop a code in C++, because this programming language has no true object code convention, because it is a object extended language and not a true object language. You just have to see names of STL classes, written lower-case, as C types to understand the difference between object languages such as Java and the modelling language UML.

This is the reason why we will have conventions in the mainframe of the XLiFE++ project : a naming convention, a code convention, and a documentation convention. The last one is dedicated to standardize comments, in order to use Doxygen.

[1] http://www.loribel.com/java/normes/intro.html (fr)

2 Naming convention

2.1 General points

- The code reference language is English.
- User objects and only user objects can be aliased in other languages than English.
- An acronym used in the name of a class, function, variable, ... behaves like a word. It is written lowercase except the first letter in specific cases we will see in the following.
- Explicit abbreviations are strongly encouraged.
  Ex : Ref for Reference, Geom for Geometry, Elem for Element, Nb for Number, ...
- **The encoding of every file is UTF-8**

2.2 Directories

Directories behave like libraries. That is evidence for Java programmers. Their names are defined in the same way.

1. A directory name is written with words joined without delimiters. Each word is capitalized except the first one.
   Ex : directSolvers
2. The _ symbol is forbidden.

2.3 Files

1. The encoding of every file is UTF-8
2. A header file extension is .hpp
3. A source file extension is .cpp
4. The name of a file that defines at least one class is the same as the main class defined in it. As a result, the name is written with capitalized words joined without delimiters.
   Ex : MsgFormat.hpp
5. The name of a file that defines only external functions is written with capitalized words joined without delimiters, except the first word that is lower-cased.
   Ex : stringUtils.cpp

2.4 Class, enum, struct and typedef

In the following, we will not make differences between the four types, unless specific mentions.

1. The name of a class is written with capitalized words joined without delimiters.
   Ex : `GeomRefElemSide`
2. The name of a typedef on a class is written with capitalized words joined without delimiters. It is a short name remembering the original class name with the suffix _t.
3. The name of a typedef on a variable is written with capitalized words joined without delimiters except for the first one that is lower-cased. It is a short name remembering the original variable name with the suffix _t.
4. An enum item is written with capitalized words joined without delimiters except for the first one that is lowercased, with the prefix _
5. The _ symbol is forbidden except for the typedefs and the following cases :
   (a) For a template class : if a template parameter is devoted to be an iterator, you will add the suffix _it.
6. If a class is dedicated to be a collection or if it inherits of a collection from the STL, you will use the plural and not the suffix List.
   Ex : `ParameterList` $\longrightarrow$ `Parameters`

2.5 Functions

1. The name of a function is written with capitalized words joined without delimiters, except the first word that is lowercase.
   Ex : `stringUtils`
2. The _ symbol is forbidden.

2.6 Variables

1. The name of a variable is written with capitalized words joined without delimiters, except the first word that is lowercase.
   Ex : `refElements`
2. The _ symbol can be used only in the following cases :
   (a) For a private or protected attribute, except for pointers, you will add the suffix _.
   Ex : `point_`
   (b) For a pointer, public or not, you will add the suffix _p.
   Ex : `parentDomain_p`
   (c) For an iterator, you will add the prefix it_ if you want to give a name different from "it". You may use the variable name to which the iterator is defined.
   Ex : an iterator on a Matrix object named "a" should be called it_a.
3. If a variable is a collection, you will use the plural instead of the suffix List.
   Ex : `elementList` $\longrightarrow$ `elements`

2.7 Macros

1. The name of a macro is written with uppercase words joined with the delimiter _.
   Ex : #define DEBUG_ON

2. For macros dedicated to multi-include protection, the name is based on the file name and a _ replaces the dot.
   Ex : #define GEOM_REF_ELEMENT_SIDE_HPP for GeomRefElementSide.hpp

3 Code convention

3.1 General points

1. The reference language is English.
2. User objects and only user objects can be aliased in other languages than english.
3. **In any case, you will use exceptions functionalities.** You use the messages manager developed in the code.
4. **In any case, user classes are template classes.** The main program has to be the easiest it is possible.
5. Each piece of code is protected by a namespace defined in header files. The namespace is xlifepp.
6. **In any case, you will use the `using` keyword in a header file**. You can use it in source files. Then, the user has the choice in the main program to use it or not.

3.2 Files

1. When you define a class, you define a header file (extension .hpp) and a source file (extension .cpp). 2 cases may occur :
   (a) the header file is included only by other header files from the same library (i.e. the same directory). In this case, there is nothing more to do, it is an "internal header file".
   (b) the header file is included by headers from others libraries (i.e. other directories). In this case, you define a header file with the same name, .h as extension, place it in the headers directory, and include the original header file (extension .hpp)

   When you compile your class, you will only need what is in the library you are in and what is in the headers directory. This is a way to hide the file organisation to each piece of the code.

3.3 Classes, enums, structs and typedefs

1. It is highly recommended to use the pointer on current object **this** inside class operations to avoid ambiguities between external functions and class operations of the same name. An external developer will have better and quicker access to the understanding of your piece of code.

3.4 Loops and tests syntaxes

1. Only the 2 following forms are authorized :
   (a) The one-line syntax, with or without braces.
   Ex :

```
if (n>0) { a(n) = 1; }
```

(b) The multi-lines syntax with braces, whatever their position
Ex :

```
for (int i=0; i<n; i++) {
  a(i)=1;
}
for (int j=0; j<n; j++)
{
  b(j)=2;
}
```

4 Documentation convention

Before showing the convention itself, we will introduce briefly how to use DOXYGEN and its way of thinking in a source code.

4.1 Brief introduction to DOXYGEN

4.1.1 DOXYGEN, how does it work ?

Documentation blocks managed by DOXYGEN are comments blocks with special markings (different from // and /* ... */) and special macros.
Almost every code comments in MELINA++ are incompatible with authorized markings and syntaxes of DOXYGEN. This is the reason why we have to normalize documentation blocks by choosing the most flexible syntax according to our needs.

Furthermore, the documentation generator can be configured with the Doxyfile file. I will not detail the content of this file and I propose you to go and see the DOXYGEN doc if you want to learn more about it.
However, a Doxyfile file will be available for those who want to test their code and documentation before the validation step in the frame of this project.

The adopted syntax is based on the Qt style, that explains some of the parameters used in Doxyfile.

4.1.2 Concepts of brief / detailed comments

A brief comment is a comment written with a unique sentence, preferentially on a unique line. It is used directly to define the piece of code it refers. For example, a brief comment associated to an attribute will be shown in the description of attributes list of a class, as in the following :

```
class A {
  public :
    //! brief comment associated to attribute i
    int i;
    int j; //!< brief comment associated to attribute j
    /*! brief comment associated to attribute k */
    int k;
    int l; /*!< brief comment associated to attribute l */

  //! brief comment associated to function f1
   void f1();
   void f2(); //!< brief comment associated to f2
};
```

In this example, you can notice the different syntaxes according to its place, before the documented element or after (just after ; in fact). The < symbol is introduced in the DOXYGEN doc as an arrow pointing to the documented element.

A detailed comment is a multi-sentences and multi-lines comment, and event a multi-paragraph comment. The default syntax is the following :

```
class A {
  public :
```

```
    /*!
      detailed comment
      associated to attribute i
     */
    int i;
    /*!
      detailed comment
      associated to function f
     */
   void f();
};
```

There are several ways to define brief and detailed comments simultaneously. The easiest way is tu use a DOXYGEN parameter which activates the following behaviour : the first sentence of a detailed comment is taken as brief comment. So you write the brief comment, you skip a line and you write the detailed comment. The skipped line is just a way to highlight the brief comment. No macro is used in this solution. From a technical point of view, the concerned parameter in Doxyfile is QT_AUTOBRIEF = YES.

4.1.3 Grouped comments

You can document several elements of a code (attributes or functions) with the same comment. You just have to define a group containing those elements.

There are several ways to define groups in DOXYGEN. The easiest way, with the least letters to use, is the following :

```
////! shared commentvoid f(int);void f(double);void f(float);void f(string);//}
```

4.1.4 To define a list

In a detailed comment, you may wish to use a list. The syntax is the following :

```
class A {
  public :
    /*!
        first part of comment associated to i
        - item1
        - item2

        next part of comment associated to i
     */
    int i;
};
```

You need **to skip a line after the last item** to tell to DOXYGEN that the last item is ended.
For a numbered list, bullets are -#.
For nested lists, indentation determines which list you are in. A tabulation of 4 spaces has to be respected :

```
class A {
  public :
    /*!
        first part of comment associated to i
        - item1
            - subitem1
            - subitem2
        - item2

        next part of comment associated to i
     */
    int i;
```

```
};
```

4.1.5 To insert code or diagrams

When you want to insert source code in your comment, or ASCII diagrams (for example, the numbering of the tetrahedron P2), you have to tell DOXYGEN to consider spaces as characters. So, you will use the DOXYGEN environment \verbatim - \endverbatim, according to the following syntax :

```
/*!
    first part of comment associated to A

    \verbatim
    source code or text diagram
    \endverbatim

    next part of comment associated to A
 */
class A {
  public :
    /*!
        detailed comment associated to attribute i
     */
    int i;
    //! detailed comment associated to signature of f
   void f();
};
```

4.1.6 Comments ignored by DOXYGEN

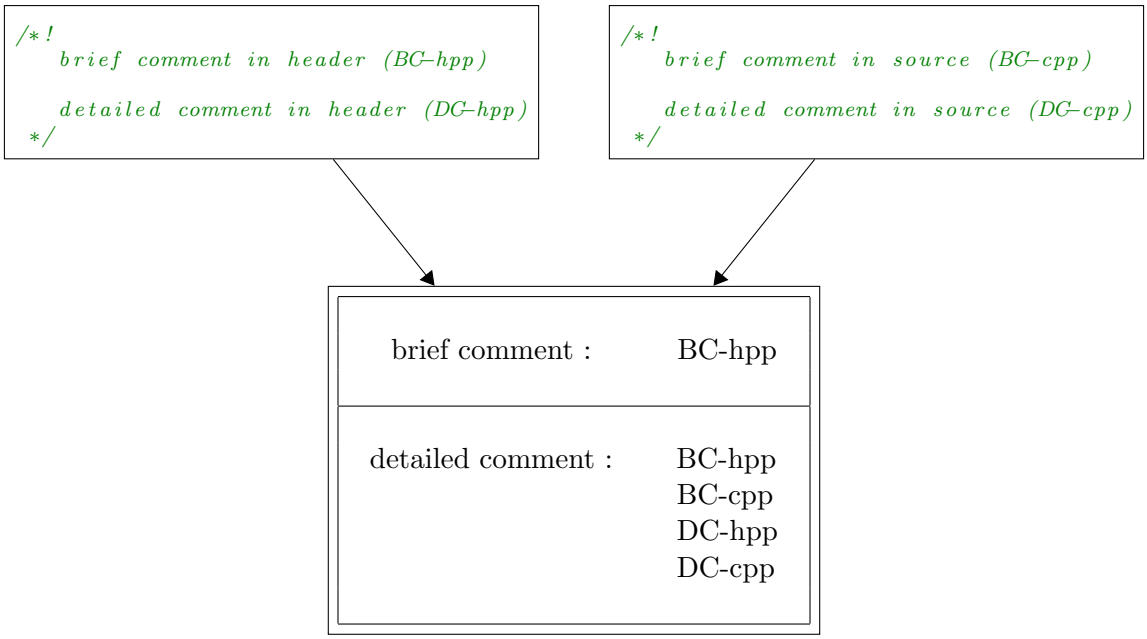
Standard C++ comments are ignored by DOXYGEN. If you want to decorate such comments, you can use one of the following box :

```
...
//////////////////////////////////////////////////
// comment in a box
//////////////////////////////////////////////////
...
//================================================
// other comment
// in a box
//================================================
...
```

Nevertheless, this decoration concerns only ignored comments.

4.1.7 Brief / detailed comments and header / source files

There is now a question to ask : How does DOXYGEN work when the declaration and the definition of a function are both documented ?



The understanding of this mechanism allows us to give the first rules of the documentation convention.

4.2 General points

1. The reference language of the documentation is English.
2. Function or operation declarations are documented with brief (one-line, one-sentence) comments and only brief comments.
3. Function or operation definitions (inline or not) are documented by detailed comments.

4.3 Documentation of a file

Here is the structure of the comment you have to write to document a file (header or source) :

```
/*!
    brief comment

    \file Parameter.hpp
    \author T. Anderson
    \date 25 jan 2011
    \since 3 mar 2007

    next part of detailed comment
 */
```

This comment is dedicated to present the content of the file : list of classes, list of external functions, ... It is written at the top of the file before everything else.

Some additional remarks :

- Thanks to the QT_AUTOBRIEF = YES parameter, the first line of the comment is interpreted as the brief comment. Therefore, the brief comment has to be placed before everything else. There is another solution with the macro \brief :

```
/*!
    \file Parameter.hpp
    \author T. Anderson
    \date 25 jan 2011
    \since 3 mar 2007

    \brief brief comment

    next part of detailed comment
 */
```

- The \date macro is the last modification date
- The \since macro is the file creation date.
- When there are several authors, you will use the \authors macro :

```
/*!
    brief comment

    \file Parameter.hpp
    \authors T. Anderson, R. Cypher
    \date 25 jan 2011
    \since 3 mar 2007

    next part of detailed comment
 */
```

In this case, authors are separated by commas.

The file documentation rules are :

1. The file documentation is written at the top of the file, before everything else.
2. The file documentation follows one of the previous syntaxes, with the 4 pieces of information file name, author(s), creation date and last modification date.

4.4 Documentation of a class, a typedef, an enum or a struct

1. A class documentation is written just before the class declaration/definition in the header file.
2. The structure of the class documentation is as follows :

```
/*!
    \class A
    brief comment associated to A

    next part of detailed comment associated to A
 */
class A {
...
};
```

3. To document a typedef, you will use the \typedef macro instead of the \class macro.
4. To document a struct, you will use the \struct macro instead of the \class macro.
5. To document an enum, you will use the \enum macro instead of the \class macro.

4.5 Documentation of attributes

1. Attributes can be documented according to one of the following syntax :

```
class A {
  public :
    //! comment associated to i
    int i;
    double j; //!< comment associated to j
    /*!
      comment associated to k
     */
    B k;
    ...
```

4.6 Documentation of functions and constructors

1. Functions and operations declarations are documented with a brief comment, and only a brief comment. The syntax, shown here for operations but valid for external functions, is one of the following :

```
class A {
  public :
    ...
   //! brief comment associated to declaration of f
   void f();

   /*!
       brief comment associated to declaration of g
    */
   void g();

   void h(); //!< comment associated to declaration of h
};
```

2. Functions and operations definitions (inline or not) are documented with detailed comments. If the declaration has not been commented first, the first line of the comment will be the brief comment, as seen in previous sections. The syntax, shown here for inline operations but valid for the others types, is one of the following :

```
class A {
  public :
    B b;
    ...
   //! comment associated to definition of f
   void f() {
   ...
   }
   void g() //! comment associated to definition of g
   {
   ...
   }
   A() //! comment associated to definition of composite constructor
   : B()
   {
   ...
   }

   /*!
```

```
      comment associated to definition of h
    */
   void h() {
   ...
   }
};
```

You may notice that for the function g, the comment marking is //! and not //!< and that for the constructor in the same case, the comment is placed before components constructor calls.

Let's see how to document function arguments.

4.7 Documentation of arguments

1. In the case only of a function definition (or an operation definition), you may want to document arguments and return value. The syntax is one of the following :

```
/*!
    comment of function f
 */
int f (double d, //!< comment of d
       int i //!< comment of i
      ) {
...
}
/*!
    comment of function g
    \param d comment of d
    \param i comment of i
    \return comment of return value
 */
int g (double d, int i) {
...
}
```

5 Template files

In the previous sections, you saw how to document a function. Let's see the templates files for header files and source files you will have to respect:

5.1 Template header file

```
/*!
    \file MyFile.hpp
    \author T. Anderson
    \date 25 nov 2011
    \since 22 nov 2011

    \brief brief comment

    detailed comment
 */

#ifndef MY_FILE_HPP
#define MY_FILE_HPP

#define OTHERS_MACROS

//=====================================
#include "code header"
...
#include "system header"
...

namespace xlifepp {

//=====================================
global stuff and fake declarations
//=====================================

/*!
    \class A
    brief comment

    detailed comment
 */
class A {
...
};
...
declaration / definition of external functions using class A
...
//=====================================

/*!
    \class MyFile
    brief comment

    detailed comment
```

```
 */
class MyFile {
...
};
...
declaration / definition of external functions using class MyFile
...
} // end of namespace xlifepp

#endif /* MY_FILE_HPP */
```

5.2 Template source file

```
/*!
    \file MyFile.cpp
    \author T. Anderson
    \date 26 nov 2011
    \since 22 nov 2011

    \brief brief comment

    detailed comment
 */

#include "MyFile.hpp"
#include "code header"
...
#include "system header"
...

using namespace std;
...

namespace xlifepp{

functions definitions using class A
...

functions definitions using class MyFile
...
}
```