Coding Standard

Table of contents

1. General

- The language for code is English. This applies for all names and identifiers in the code as well as for all comments.
- Don't use C-style casts.
- Class definitions and their implementations are organized in different files (.h and .cpp). It is not allowed to implement the class in the header file. Each class has its own pair of header and cpp files. Exceptions to this rules are:
- Small classes which semantically belong together. E.g. All exceptions of one module should go in one file.
- Templates (This is due to compiler limitations, if compilers get better in the future, this rule might get obsolet)
- Prefer initialization to assignment of members.
- Public methods have always to be checked against pre- and post-conditions as well as invariants.
- Debugging code (e.g. checks for pre-/postconditions) has to be surrounded by appropriate macros so that they can be removed, when compiling final releases. For removing the debug code in release versions the macro NDEBUG should be used.Please consider using existing macros like assert.

2. Naming

Name identifiers according to the following naming conventions:

- Namespaces: AllWordsCapitalizedWithoutSeparatingButTrailingUnderscore _
- Classes: AllWordsCapitalizedWithoutUnderscores
- Methods: firstWordLowerCaseRestCapitalizedWithoutUnderscores
- Constants: ALL_UPPER_CASE_WITH_UNDERSCORES
- Class and instance member variables: all_lower_case_with_underscores_and_with_trailing_underscore_
- Local variables: all_lower_case_with_underscores
- Exceptions: ClassNameEndsWithException

Besides these general naming rules some special naming conventions apply:

- All methods which change properties of classes should be named setXXX. The methods returning the value of certain properties of classes can be divided into two categories: getXXX for non-boolean properties and isXXX for boolean values respectively. Example:

```
void setCounter ( int value ) { . . . } int getCounter ( ) { . . . }
const
```

```
void setReadOnly (bool value ) { . . . } bool isReadOnly ( ) { . . . }
const
```

- Asterisks denoting pointers are always bound to the type name rather than to the variable name. e.g. char* my_c_string; instead of char *my_c_string;
- All headers have to be protected against multiple inclusions by appropriate preprocessor macros. The corresponding #define has to fulfill the following convention:

```
#define FILE_NAME_
```

3. Structural

- The maximum line-length is 95 characters.
- Write curly braces around code-blocks in lines of their own.
- Indent code-blocks by two spaces. Don't use tabs for indentations but use spaces instead.
- Only indent the code block, not the curly braces!
- When invoking methods the opening brace always should follow the method name without any whitespaces. The first argument follows the opening brace without a space whereas argument-separating commas are always followed by spaces. e.g:

```
my object.myMethod(argument1, argument2, argument3);
```

- Constructors, operators, methods, and variables should be grouped according to their access specifier

```
class Example
{
public :
// constructors and operators
// methods
// variables
protected :
// constructors and operators
// methods
// variables
private :
// constructors and operators
// methods
// variables
};
```

- Preceed every method declaration/definition with a comment line consisting of a double backslash and 50 - to make the code more readable.
- Every class, function and constant should belong to a namespace.
- Using directives (e.g. using namespace std;) are a source of naming conflicts and should therefore be avoided. Prefer using declarations (e.g. using std::cout)

- Every kind of using (i.e. declarations as well as directives) is strictly forbidden within header files. Direct scoping has to be used instead.
- Minimize includes in header files. Use forward declarations instead.
- Always prefer const to #define. If you intent to define a collection of integral constants use enum.
- Using declarations should be written in the following order:
  - Own classes.
  - C++ Standard library
  - Third party APIs.
- To increase the readability of the using part, all using declarations should be sorted by namespaces, that means classes, etc. belonging to the same namespace can be found in consecutive lines. Between the three categories mentioned above a single blank line is recommended.
- The const specifier is always written left-binding. e.g. int const a number instead of const int a number.
- Prefer class-local typedefs to globally defined types.
- Never use hard coded; constants except for strings describing errors within exceptions.
- Public non-const members are strictly forbidden. Use access methods instead.

4. Documentation

- Avoid the use of block comments (/* ... */) in the source code and use the line comments (// ...) instead. This makes the source code less fragile to erroneous deletions of code-lines. Please note that comments which should be available in the doxygen documentation should start with three slashes e.g. /// ... . When you precede a class definition or a method declaration/definition with a comment line of minuses please use two slashes since we do not want these lines to appear in the doxygen documentation.
- If it is absolutely necessary to put comments in the code to describe algorithmic details preceed the according code fragment with a comment block rather than spreading the comments across the code fragment.
- If it is absolutely necessary to clarify non-obvious code write short comments at the end of the appropriate code line. Nevertheless whenever such a comment seems necessary think twice if there is a better obvious solution that doesn't need a comment!
- When describing a design pattern the name of the book which deals with this pattern should be cited (e.g. [Gamma et al. 1998]).
- When describing algorithms the names of the book which deals with these algorithms and datastructures should be cited (e.g.[Sedgewick 1992]).
- Class-definitions should be preceded by a Doxygen header of the following form:

```
//---------------------------------------------------
/// ExampleClass
```

```
///
/// Description of the class. If a useful link can be given in the
running text use
/// @link fully::qualified::classname#method(fully::qalified::params)
///
/// @author <authorname and email>
///
/// @version <version > ( s tar t ing with 1 . 0 . 0 )
/// @see <fully::qualified::classname#method(fully::qualified::params)>
/// @invariant <things that must not change>
/// @deprecated <if applicable write reason here , otherwise omit this
line>
class ExampleClass
{ . . .
};
```

Please note that the preceding line of minus starts with two slashes which causes that this line does not appear in the doxygen documentation. In the description text a usage example is highly recommended. Always describe if the class is meant to be used polymorphic.

- Prefix methods by a Doxygen header of the following form (only in header files):

```
//--------------------------------------------------
/// Description of the method in HTML format , if a link can be given in
the
/// running text do this with
/// @link
fully::qualified::Classname#methodName(fully::qualified::Paramclass)
///
/// @return ReturnType
/// @param param name Descr ipt ion of the parameter .
/// @pre Precondition which should be met . Plese use only one @pre tag
and
/// write logical operators to combine them. If possible these
conditions
/// should be writ ten in c++. E. g . : @pre X. x == true AND X. y == f
a l s e
/// @post The same rul e s as for @pre apply for the post&-condi t ion .
/// @exception MySpecialException Describe when this exception is thrown
.
/// @exception AnotherSpecialException Use one @exception tag for each
/// exception
ReturnType myFunction(ParamType param name)
throw(MySpecialException , AnotherSpecialException )
{ . . .
}
```

In the description a usage example is highly recommended.

- If bad hacks are absolutely unavoidable for whatever reason (e.g. absolutely have to meet a deadline, etc..) they should be tagged by a hack-start and hack-end comment of the following form:

```
// FIXXME (<author , date>) <desciption of the hack>
[ . . . . . the hack . . . . . ]
// END FIXXME (<author , date>)
```

To facilitate a grep, the keyword FIXXME should be written in upper-case and with at least two Xs. More than two are allowed and should be used for really bad hacks - as a rule of thumb: the more Xs the word FIXXME contains the worse the hack is, up to a maximum of five Xs. All hacks that have found their way into the code should be removed as soon as possible!

5. KDevelop file templates

You can set these globally under 'Settings -> Configure KDevelop -> File Templates' or for a single project under 'Project -> Project Options -> File Templates'.

The template for header files:

```
//---------------------------------------------------
/// @file $MODULE$.h
///
/// @brief
///
/// @author Author Name (email)
/// @bug No known bugs.
//----------------------------------------------------
```

The template for implementation files:

```
//---------------------------------------------------
/// @file $MODULE$.cpp
///
/// @brief Implementation of $MODULE$.h
///
/// @author Author Name (email)
/// @bug No known bugs.
//----------------------------------------------------
```