

xmas-design coding style

A lot of successful software projects like Linux [1], boost [2], or Qt [3] have their own coding style guide. A coding style guide improves code quality, readability, and serves as a guideline for new developers.

Every file in xmas-design will sooner or later be ported by applying the guidelines described in this document. Every developer should at least consider the points made here. And sometimes, there may be a good reason to break a rule.

// not true yet: Some of the rules are already automatic tested by Krazy [6] every day.

References:

- [1] <http://lxr.linux.no/linux/Documentation/CodingStyle>
- [2] <http://www.boost.org/development/requirements.html>
- [3] <http://doc.trolltech.com/qq/qq13-apis.html> and <http://trolltech.com/developer/resources/notes/naming>
- [4] http://techbase.kde.org/Policies/Kdelibs_Coding_Style
- [5] http://techbase.kde.org/Policies/Licensing_Policy
- [6] Krazy - <http://www.englishbreakfastnetwork.org/>
- [7] http://techbase.kde.org/Policies/Library_Code_Policy
- [8] astyle - <http://astyle.sourceforge.net/>
- [9] apidox - http://techbase.kde.org/Development/Tutorials/API_Documentation

Filename

- We use lowercase filenames like `mainwindow.h` and `mainwindow.cpp`

Variable names

- Normal variables are generally lowercase
- class names are camel case (starting with uppercase) like `CamelCase`
- class names in variables are camel case except for the first character like `camelCase`.
- class member name variables start with `m_`.
- class member name getters and setters have the same name as the member except for the `m_`.

```

class CamelCase {
    private:
        string m_member_name;
    public:
        const string member_name();           // getter
        void member_name(string member_name); // setter
}

int main(int argc, char **argv) {
    CamelCase camelCase = new CamelCase();

    string copy = camelCase.member_name();
}

```

Formatting:

- Indentation

- 1.) Do not use tabs [4].
- 2.) Indentations are 4 characters deep [4].

- Whitespace

- 3.) Do not leave whitespaces at the end of lines.
- 4.) Use blank lines to group statements [4].
- 5.) Use only one empty line to separate items [4].
- 6.) Use one space after each keyword [4].

Example:

```

if(    // wrong
if (  // correct

```

- 7.) Usage of whitespaces between an opening and closing parenthesis (e.g. if- and for-statments, function calls) is up to the developer.

Example:

```

if (i < 5)                                <->  if ( i < 5 )
calculateSalary(age, years);               <->  calculateSalary( age, years );

```

8.) For pointers or references, use either C++ style or C style. We prefer C style, because that coincides better with semantics (see beware).

So, either a single space after '*' or '&', but not before (C++ style),

Example:

`T* v` and `T& v`

Beware:

`char* i, j;` // `i` is declared pointer to char, while `j` is declared char

or a single space before '*' or '&', but not after (C style).

Example:

`T *v` and `T &v`

10.) Do not use spaces around '.' or '->', nor between unary operators and operands.

- Braces

11.) Function implementations, class, struct and namespace declarations always have the opening brace on the start of a line [4].

12.) For all other constructs (if, switch, for, ...), the left curly brace goes on the same line as the start of the statement [4].

13.) Use curly braces even when the body of a conditional statement contains only one line [4].

- Statements

14.) Do not put multiple statements on a single line.

- Switch statements:

15.) Case labels are on the same column as the switch [4].

- Line breaks

16.) Try to keep lines shorter than 100 characters, inserting line breaks as necessary [4].

- Pointers

17.) In C++, a null pointer is `nullPtr` (not any of the 0-literals).

Variable declaration

- 18.) Each variable declaration on a new line [4].
- 19.) Variables and functions start with a lowercase letter.
- 20.) Classes always start with a capital letter and have CamelCase names.
- 22.) Member variables start with “m_”.
- 23.) Do not use pseudo Hungarian style.

Example:

```
m_bSomeBoolean, m_pSomePointer // wrong
m_someBoolean, m_somePointer   // correct
```

- 24.) Variables (objectName) in ui files start with “ui_”.

Rationale:

This makes it possible to identify the source of the variable (defined in class or in ui file).

- 25.) Use vertical alignment to ease scanning of declarations.

Example:

```
QString  aStringToUse;
int      anInt;
double   aDoubleNumberToUse;
```

- 26.) Use static const variables instead of defining integers or floats. i.e. avoid a macro DEFINE.

Rationale:

This provides type-safety.

Casting:

- 27.) Use C++ style cast like `static_cast`, `dynamic_cast` instead of C style cast as it asserts the purpose of the cast. Also try to use Qt casts like `qobject_cast`, `qgraphicsitem_cast` in place of `dynamic_cast` wherever possible as the Qt style casts succeeds even across libraries. It doesn't mean dynamic casts shouldn't be used though. Use them wherever it is impossible to use Qt's cast.

License:

28.) All source code and related data files must have the following license header from the first line on of the file (see also [5])

```
/*
Copyright <year>  xmas-design
// we don't have a central email address Authors <xmas-design-devel@kde.org>

This program is free software; you can redistribute it and/or modify it
under the terms of the GNU General Public License as published by the Free
Software Foundation; version 3 or any later version

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program.  If not, see <http://www.gnu.org/licenses/>.
*/
```

Includes:

29.) Include own header first

30.) Include Qt QClasses in angle brackets (checked by Krazy [6]). But do not add the QtModule like QtGui, because this will change in Qt5 and compilation will break.

Example:

```
#include <QtGui/QMenu>  // wrong
#include <QMenu>         // correct
```

32.) Do not add a path to the include statement.

Example:

```
#include "../uml.h"  // wrong
#include "uml.h"     // correct
```

Rationale:

Paths are set in the make files. Rearranging or moving files should not be followed by source code editing.

33.) Header includes should be listed in the following order and grouped

- own header
- xmas-design includes
- Qt includes

34.) The headers inside each group should be sorted.

Rationale:

For ease of locating them.

Tip:

Kate/KDevelop users can sort the headers automatically.
Select the lines you want to sort, then
Tools -> Filter Selection Through Command -> "sort".
In vim the same can be achieved by marking the block, and then
doing ":sort".

In emacs, you can mark the block and then do "M-x sort-lines".

35.) Includes in a header file should be kept to the absolute minimum, as to keep compile times low. This can be achieved by using forward declarations.

Forward declarations work for pointers and const references.

Include guards:

36.) Include macro characters are all in uppercase letters.

Example:

```
#ifndef MyFileName_h    // wrong
#define MY_FILE_NAME_H // correct
```

37.) Do not use leading or trailing underscores on the include guard macro as they are reserved for compiler/libc use.

Example:

```
#ifndef _MY_FILE_NAME_H_ // wrong
#define MY_FILE_NAME_H   // correct
```

Doxygen comments:

38.) Every public item must have a doxygen comment.

39.) Doxygen comments look like (see also [9]):

```
/**
 * Here is the description...
 * @param ...    ...
 * @return      ...
 */
```

40.) The comments for methods should be written in the implementation file (cpp). The comments for the class should be written in the header file.

Rational:

The header files can be overviewed easier and read quicker.

Header structure:

41.) According to policy, a C++ header file should contain only 1 publicly visible class.

42.) Use the following layout:

```
<license>
<include guard>
<includes>

namespace
{

/**
 * description
 */
class <name> : public <parent>
{
public:

protected:

private:

};
}
```

C++:

- 43.) “const correctness” should be preserved as much as possible. Make all getters const.
- 44.) It might be a good idea to make the constructors explicit

Metrics

- loc = lines of code
- Refactor classes, which have more than 1000 loc per class.
- Refactor methods, which have more than 200 loc per method.

Astyle

You can use `astyle` to format one or several files with the following command:

```
astyle --indent=spaces=4 --brackets=linux \  
      --indent-labels --pad=oper --unpad=paren \  
      --one-line=keep-statements --convert-tabs \  
      --indent-preprocessor \  
      'find -type f -name '*.cpp' ' 'find -type f -name '*.h' '
```

or

```
astyle --indent=spaces=4 --brackets=linux \  
      --indent-labels --pad=oper --unpad=paren \  
      --one-line=keep-statements --convert-tabs \  
      --indent-preprocessor \  
      < Original > Beautified
```

This applies some of the rules in section ‘Formatting’.