Radiy Department C, C++ Style Guide, Qt Framework

Version 1.6, 27 December 2016

**Table of Contents**

[1. General 1](#_Toc470598594)

[2. Header Files 1](#_Toc470598595)

[3. Scoping 2](#_Toc470598596)

[4. Classes 3](#_Toc470598597)

[5. Functions 5](#_Toc470598598)

[6. Other c++ features 5](#_Toc470598599)

[7. Naming 7](#_Toc470598600)

[8. Comments 8](#_Toc470598601)

[9. Formatting 8](#_Toc470598602)

[10. 8](#_Toc470598603)

# General

These rules are applied for developing projects using Qt Framework. Coding style and formatting are pretty arbitrary, but a project is much easier to follow if everyone uses the same style. Individuals may not agree with every aspect of rules, and some of them may take some getting used to, but it is important that all project contributors follow the style rules so that they can all read and understand everyone's code easily.

When strictly following a rule makes code look bad, it is allowed to break it.

Note that older code might not conform to these rules and should be fixed when convenient or should be left as is.

# Header Files

In general every .cpp file should have an associated .h file, non-header files that are meant for inclusion should have an extension .inc. All headers should have a header guard.

## Header Guard

All header files should have #define or #pragam once guard to prevent multiple inclusions. The format for #define guard is default QtCreator style.

#ifndef TYPESANDENUMS\_H

#define TYPESANDENUMS\_H

...

#endif // TYPESANDENUMS\_H

## Inline Functions

Functions can be defined as inline only when they are small, just several lines. Inlining a function can generate more efficient object code, as long as the inlined function is small. Feel free to inline accessors and mutators, and other short, performance-critical functions. It is important to know that functions are not always inlined even if they are declared as such; for example, virtual and recursive functions are not normally inlined.

It is allowed to format inline function in a single line

int counter() const { return m\_counter; }

## Order of includes

Use standard order for readability and to avoid hidden dependencies: Related header, C library, C++ library, Qt Framework, other libraries' .h, your project's .h.

#include ʺStable.hʺ // Precompiler header if used

#include ʺBuilder.hʺ // Related header

#include <map> // c++ headers

#include <vector> //

#include <QWidget> // Qt headers

#include <QBuffer> //

#include ʺBuilder.hʺ // Project headers

# Scoping

## Namespaces

All library code should be placed namespace. Namespaces should have unique names based on the project name, and possibly its path.

Do not use using-directives

using namespace std; // Bad, this pollutes the current namespace

Do not use nested namespaces

namespace OuterNamespace

{

namespace NestedNamespace // Bad, makes user use it as

{ // OuterNamespace::OuterNamespace::Type

}

}

Use one tab intend inside namespace

namespace OuterNamespace

{

class Item // Ok, intend inside namespace is one tab

{

};

class Rect : public Item // Bad, no intend

{

};

}

## Local Variables

Place a function's variables in the narrowest scope possible, and initialize variables in the declaration. C++ allows you to declare variables anywhere in a function but local variables should be declared in as local a scope as possible, and as close to the first use as possible. This makes it easier for the reader to find the declaration and see what type the variable is and what it was initialized to.

In particular, initialization should be used instead of declaration and assignment, e.g.:

int i;

i = f(); // Bad – initialization separate from declaration

int i = f(); // Good – declaration has initialization

Prefer initializing using brace initialization

std::vector<int> v0;

v0.push\_back(1); // Prefer initializing using brace initialization.

v0.push\_back(2);

std::vector<int> v1 = {1, 2}; // Good -- v starts initialized

Variables needed for if, while and for statements should normally be declared within those statements, so that such variables are confined to those scopes. E.g.:

while (const char\* p = strchr(str, '/'))

{

str = p + 1;

}

There is one caveat: if the variable is an object, its constructor is invoked every time it enters scope and is created, and its destructor is invoked every time it goes out of scope.

// Inefficient implementation

//

for (int i = 0; i < 1000000; ++i)

{

Foo f; // My ctor and dtor get called 1000000 times each.

f.DoSomething(i);

}

It may be more efficient to declare such a variable used in a loop outside that loop:

Foo f; // My ctor and dtor get called once each

for (int i = 0; i < 1000000; ++i)

{

f.doSomething(i);

}

# Classes

Classes are the fundamental unit of code in C++. This section lists the main dos and don'ts user should follow when writing a class.

## Doing Work in Constructors

Avoid virtual method calls in constructors, and avoid initialization that can fail if you can't signal an error (such as file open).

Use explicit functions such as open() or init()).

Using exceptions in constructors is forbidden;

MyClass::MyClass()

{

...

if (someCondition == true)

{

throw std::invalid\_argument(ʺparamʺ); // Bad idea

}

}

Prefer inclass object initialization

class MyClass

{

...

private:

int m\_count = 0; // Good

QString m\_defaultId = ʺAPPSIGNALID####ʺ; // Good

};

Prefer delegate constructor instead of init() function

class MinMax

{

public:

MinMax() :

MinMax(0, 0) {} // Delegate all work to MinMax(int, int)

MinMax(int min, int max) :

m\_min(min),

m\_max(max) {...}

};

Constructors should never call virtual functions.

MyClass::MyClass()

{

someVirtaulFunction(); // Bad idea

}

## Implicit Conversions

Do not define implicit conversions. Use the explicit keyword for conversion operators and single-argument constructors.

class MyClass

{

public:

MyClass(int value); // Bad, implicit conversion not obvious

explicit MyClass(double value); // Good, always know what’s going on

...

};

## Struct vs classes

Use a struct only for passive objects that carry data; everything else is a class.

## Access Control

Data members should be declared as private, unless they are static const. For technical reasons (testing, inheritance, optimization) data members are allowed to be protected.

## Declaration Order

A class definition should usually start with a public:  section, followed by protected:, then private:. Omit sections that would be empty.

The common class looks like:

class MyClass

{

public:

MyClass(); // Constuctors and desctructors are first

virtual ~MyClass();

public: // public, protected, and private methods

void method1();

protected:

void method2();

private:

void method3();

// Getters/setters

//

public:

int level() const;

void setLevel(int value);

const std::vector<QString>& messages() const;

std::vector<QString> messages();

// Data

//

private: // protected or private section is last

int m\_level;

std::vector<QString> m\_messages;

};

# Functions

## Parameter Ordering

When defining a function, parameter order is: inputs, then outputs.

Parameters to C/C++ functions are either input to the function, output from the function, or both. Input parameters are usually values or const references, while output and input/output parameters will be pointers to non-const. When ordering function parameters, put all input-only parameters before any output parameters. In particular, do not add new parameters to the end of the function just because they are new; place new input-only parameters before the output parameters. Avoid using parameters which are input and output at the same time.

## Reference Arguments

All parameters passed by reference must be labeled const. Input arguments should be values or const references/pointers while output arguments are almost always pointers.

## Override Specifier

Use override to claim that a virtual function overrides another virtual function. The compiler knows that it is an override, so it can check that the user is not altering/adding new methods that user thinks are overrides.

class Derived : public Base

{

...

virtual void someOverridedVirtualFunc() override; // Correct

virtual void someOverridedVirtualFunc(); // Wrong, can declare a new

// function accidently

};

# Other c++ features

## Smart pointers

Smart pointers prevent most situations of memory leaks by making the memory deallocation automatic. More generally, they make object destruction automatic: an object controlled by a smart pointer is automatically destroyed (finalized and then deallocated) when the last (or only) owner of an object is destroyed, for example because the owner is a local variable, and execution leaves the variable's scope. Smart pointers also eliminate dangling pointers by postponing destruction until an object is no longer in use.

Prefer user of unique\_ptr where is exactly one owner of the underlying pointer.

Use reference-counted smart pointer shared\_ptr when necessary to assign one raw pointer to multiple owners.

To allocate memory use specialized functions

// Wrong, two memory allocations for reference counter and another one for Object

//

std::shared\_ptr<Object> object(new Object(param1, param2);

// Correct, single memeory allocation for reference counter and object

//

std::shared\_ptr<Object> object = std::make\_shared<Object>(param1, param2);

Never use std::auto\_ptr. Instead, use std::unique\_ptr.

## Exceptions

In most cases exceptions are not expected to be in source code. Most functions should return explicit result. However there is no strict rule to forbid using exceptions.

## Casting

Do not use C-style casts. Instead, use C++-style casts when explicit type conversion is necessary.

// Wrong, c-cstyle cast – can cast anythig to anything

//

int i = (int)someDouble;

Base\* ptr = (Base\*)object;

// Right, c++ style cast

//

int i = static\_cast<int>(someDouble);

Base\* ptr = static\_cast<Base\*>(object);

## Use of const

Use const whenever it makes sense.

Declared variables and parameters can be preceded by the keyword const to indicate the variables are not changed (e.g., const int foo). Class functions can have the const qualifier to indicate the function does not change the state of the class member variables (e.g., class Foo { int Bar(char c) const; };).

const variables, data members, methods and arguments add a level of compile-time type checking; it is better to detect errors as soon as possible. It is strongly recommended using const whenever it makes sense to do so:

If a function guarantees that it will not modify an argument passed by reference or by pointer, the corresponding function parameter should be a reference-to-const (const T&) or pointer-to-const (const T\*), respectively.

Declare methods to be const whenever possible. Accessors should almost always be const. Other methods should be const if they do not modify any data members, do not call any non-const methods, and do not return a non-const pointer or non-const reference to a data member.

Consider making data members const whenever they do not need to be modified after construction.

The mutable keyword is allowed but is unsafe when used with threads, so thread safety should be carefully considered first.

## 0, 0.0 and nullptr

Use 0 for integers, 0.0 for reals, nullptr for pointers.

## Lambda Expressions

Use lambda expressions where appropriate, prefer the explicit captures.

std::find\_if(v.begin(), v.end(),

[&val](const object& x) // val is capturead by ref, don’t use [&] or [=]

{

return object.val == val;

});

## C++11, C++14

Use libraries and language extensions from C++11(14) when appropriate. Consider portability to other environments before using C++11, C++14 or any other latest features in the project.

## Containers and Algorithms

Prefer to use standard libraries and algorithms prior to Qt or any other framework/library.

The exceptions are QByteArray and QString, QStringList and some others.

# Naming

The most important consistency rules are those that govern naming. The style of a name immediately informs user what sort of thing the named entity is: a type, a variable, a function, a constant, a macro, etc., without requiring to search for the declaration of that entity.

## General Naming Rules

Names should be descriptive. Do not use abbreviations that are ambiguous or unfamiliar to readers outside the project, and do not abbreviate by deleting letters within a word.

// Correct

//

int priceCountReader; // No abbreviation.

int numErrors; // "num" is a widespread convention.

int numDnsConnections; // Most people know what "DNS" stands for.

// Wrong

//

int nerr; // Ambiguous abbreviation.

int wgcConnections; // Only your group knows what this stands for.

int pc\_reader; // Lots of things can be abbreviated "pc".

int cstmrId; // Deletes internal letters.

## File Names

Filenames should be Pascal-case. C++ files should end in .cpp and header files should end in .h. Files that rely on being textually included at specific points should end in .inc.

Do not use filenames that already exist in /usr/include, such as db.h.

In general, make your filenames very specific. For example, use HttpServerLogs.h rather than Logs.h.

Examples of acceptable file names:

MyUsefulClass.cpp

MyUsefulClass.h

## Type Names

Type names start with a capital letter and have a capital letter for each new word, with no underscores: MyExcitingClass,  MyExcitingEnum.   
The names of all types — classes, structs, type aliases, enums, and type template parameters — have the same naming convention. Type names should start with a capital letter and have a capital letter for each new word. No underscores.

## Variable Names

Variable names start with a lowercase letter and have a capital letter for each new word, with no underscores: myExcitingVariable, superBlock.   
Data members of classes (but not structs) additionally have leading m\_.

For instance:

int count = 0; // Local variable

int m\_count = 0; // Data member of class m\_- means member

## Constant Names

Variables declared constexpr or const, and whose value are fixed for the duration of the program, are named as TypeNames, start with a capital letter and have a capital letter for each new word, with no underscores: TotalItemCount, PacketSize.

## Function Names

Function names should start with a lowercase letter and have a capital letter for each new word (Pascal case). Such names should not have underscores. Prefer to lowercase acronyms as single words (i.e. StartRpc(), not StartRPC()).

addTableEntry()

deleteUrl()

openFileOrDie()

Accessors and mutators (get and set functions) may be named like variables. These often correspond to actual member variables, but this is not required. For example, int count() and void setCount(int count).

## Macro Names

Macro names should be named with all capitals and underscores MY\_MACRO\_THAT\_SCARES\_SMALL\_CHILDREN.

#define ROUND(x) ...

#define PI\_ROUNDED 3.0

# Comments

Comments are absolutely vital to keeping code readable. The following rules describe what should be commented and where. While comments are very important, the best code is self-documenting. Giving sensible names to types and variables is much better than using obscure names that must be then explained through comments.

## Comment Style

Use either the // or /\* \*/ syntax, however, // is *much* more common

## File Comments

Start each file with license boilerplate. File comments describe the contents of a file. If a file declares, implements, or tests exactly one abstraction that is documented by a comment at the point of declaration, file comments are not required.

## Class Comments

Every non-obvious class declaration should have an accompanying comment that describes what it is for and how it should be used. The class comment should provide the reader with enough information to know how and when to use the class, as well as any additional considerations necessary to correctly use the class.

## Function Comments

Declaration comments describe use of the function (when it is non-obvious), comments at the definition of a function describe operation.

# Formatting

Coding style and formatting are pretty arbitrary, but a project is much easier to follow if everyone uses the same style. Individuals may not agree with every aspect of the formatting rules, and some of the rules may take some getting used to, but it is important that all project contributors follow the style rules so that they can all read and understand everyone's code easily.

## Tab Policy

Use tabs only, tab size is 4 spaces. Set your editor to use tabs only

## Functions Declarations

## Bracing

## Return Values

## Conditionals

Prefer no spaces inside parentheses. The if and else keywords belong on separate lines.

## Class Format

## Namespace Formatting

Скобки начала и конца блока располагаются друг под другом и с новой строки!

|  |
| --- |
| Правильно:  if (a == 0)  {  }  Неправильно:  if (a == 0) {  } |

1. После названия функции НЕТ ПРОБЕЛА

|  |
| --- |
| 1. Правильно: 2. func(param1, param2); 3. Неправильно: 4. func (param1, param2); |

1. Пробел при перечислении параметров функции ставиться только ПОСЛЕ ЗАПЯТОЙ

|  |
| --- |
| 1. Правильно: 2. func(param1, param2); 3. for (int i = 0; i < 10; i++) 4. Неправильно: 5. func( param1, param2); 6. for ( int i = 0;i < 10;i++) |

1. В циклах, условиях фигурные скобки ставить всегда

|  |
| --- |
| 1. Правильно: 2. for (...) 3. { 4. } 5. if (condition == true) 6. { 7. } 8. Неправильно: 9. for (...) 10. func( param1, param2); |
|  |

2. wdw
3. Другое
4. 5.
5. Правильно:
7. Неправильно:
8. for (;;)
9. func();
10. if (condition == true)
    * 1. func();
11. 5. Если в условии есть else обязательно должны быть фигурные скобки.
12. Правильно:
13. if (i == 0)
14. {
15. func1();
16. }
17. else
18. {
19. func2();
20. }
21. Неправильно:
22. If (i == 0)
23. {
24. func1();
25. }
26. else
27. func2();
28. 6. Комментарии писать с БОЛЬШОЙ БУКВЫ и на следующей строке ставить пустой комментарий. После // ставить пробел.
29. Правильно:
30. // Comment
31. //
32. Неправильно:
33. // comment
34. //Сomment
35. 7. Рекомендация: функции не возвращающие значение (void) должны оканчиваться явным return.
36. void func()
37. {
    * 1. …
      2. return;
38. }
39. 8. Все выражения разделять пробелом (операции пре/пост инкремента/декремента разрешается не отделять пробелами):
40. Правильно:
41. a = a + b + c;
42. a > b;
43. for (int i = 0; i < 100; i++)
44. i++;
45. i ++;
46. Неправильно:
47. a = a+b+c;
48. if (a>b) ...
49. for (int i=0; i<100; i++)
50. 8. Все объявленные переменные должны быть инициализированы (даже если дальше они на 100% будут инициализированы).
51. 9. При входе в функцию обязательно проверить все входные параметры на корректность.
52. 10. После освобождения выделенной памяти, указателю присвоить nullptr  (исключение – деструктор).
53. 11. В условиях запрещается использовать логическое отрицание (исключение – тернарный оператор).
54. Правильно:
55. if (boolVar == true)
56. if (boolVar == false)
57. if (pointer == nullptr)
58. if (pointer != nullptr)
59. if (var == 0)
60. if (var != 0)
61. Неправильно:
62. if (boolVar)
63. if (!boolVar)
64. if (pointer)
65. if (!pointer)
66. if (var)
67. if (!var)
68. 2. Переменные и функции именуются в CamelStyle
69. int count = 0;
70. void showWindows(bool show);
71. 2. Переменные члены классов должны иметь префикс m\_
72. class SomeClass
73. {
74. …
75. private:
    * 1. int m\_itemCount;
      2. std::vector<int> m\_items;
76. };
77. 12. Не использовать венгерскую нотацию (в том числе для указателей и дескрипторов).
78. Правильно:
79. int variable = 0;
80. bool isStatic = false;
81. int\* buffer = nullptr;
82. Неправильно:
83. int iVariable = 0;
84. BOOL bStatic = FALSE;
85. 14. При объявлении указателя символ «\*» ставится вместе типом а не вместе с переменой–указателем.
86. Правильно:
87. int\* someData = nullptr;
88. Неправильно:
89. Int \*someData = nullptr;
90. 16. После ключевых слов if, for, while, do ставить пробел.
91. Правильно:
92. if (a == 0)
93. for (int i = 0; i < 10; ++)
94. Неправильно:
95. if(a == 0)
96. for(int i = 0; i < 10; ++)
97. 17. Одна переменная – одна строка.
98. Правильно:
99. int index = 0;
100. QObject\* object = nullptr;
101. Неправильно:
102. int index, i, f;
103. 19. Использовать namespace
104. Правильно:
105. namespace VFrame30
106. {
107. class …
108. }
109. 20. При наследовании виртуальных функций использовать override
110. Правильно:
111. virtual void Func1() const override;
112. virtual void Func2() override;
113. Неправильно:
114. virtual void Func1() const;
115. virtual void Func2();
116. 21. При работе с файлами, обязательно выполнять:
117. проверку наличия файла;
118. проверку операций ввода/вывода (на объем считанной/записанной информации);
119. отображать ВСЕ ошибки работы с файлом (в развернутом виде, с возможными путями устранения проблемы);
120. при считывании/записи больших объемов информации выводить ход выполнения задачи;
121. определение прав доступа при создании файла;
122. отдавать предпочтение записи текстовой информации в UNICODE формате.
123. 22. Правила создания диалоговых окон
124. Диалоговое окно не должно содержать ничего лишнего и должно быть как можно проще.
125. Объединяйте элементы в логические группы, снабжая их прямоугольной рамкой и подписью (Group).
126. Если окно содержит много элементов, то постарайтесь разбить их на группы и разместить их с помощью закладок.
127. Нежелательно, чтобы закладки в диалоговом окне превышали более одного ряда, это усложняет поиск.
128. Избегайте создания диалоговых окон с неизменяемыми размерами.
129. По по возможности используйте стандартные элементы интерфейса, хорошо знакомые пользователям.
130. Установить правильный TabOrder.
131. Команды меню, вызывающие диалоговые окна, должны оканчиваться многоточием, например Open,.. (Открыть...).
132. Старайтесь не добавлять меню в диалоговые окна.