CODING STANDARDS

Deliverable 3

FEBRUARY 28, 2021

Allison Costa Amaral (N0871194)
Aurora Fernandes Freire (N0852831)
Callum Thomas (N0810466)
Rui Pereira (N0857014)

Contents

Coding Standards

Introduction

The following document has been created to illustrate the coding standards that will be implemented in our project. As mentioned in previous documents we are developing a messaging application called Focus. The application will be created in C++ using a range of libraries such as Boost and STL data structures.

These guidelines are heavily inspired on Google's Style Guide for C++.

Purpose

The purpose of this document is to enforce a standard during development to avoid future issues that can affect the success of this project's release. There are many things that can go wrong during development, creating a set of coding standards is beneficial in order to ease the likelihood of such problems occurring. Most commonly, problems range from debugging, scaling and maintainability issues.

C++ version

Code should only include features from version C++ 17.

Recently added features from newer versions should not be used and portability between versions should be considered when using features from older versions.

Header Files

Header files should be self-contained (compile on their own) and end in .h.

All header files should be self-contained. Users and refactoring tools should not have to adhere to special conditions to include the header. Specifically, a header should have header guards and include all other headers it needs.

The #define Guard

All header files should have #define guards to prevent multiple inclusion. The format of the symbol name should be <FILENAME>_H.

```
#ifndef FILENAME_H
```

Every header (.h) file should have an associated source (.cpp) file. Some exceptions are unit tests and small .cpp files.

Forward Declarations

Forward Declarations are to be avoided. Instead, include the necessary headers.

```
// b.h:
struct B {};
struct D : B {};
```

```
// good_user.cpp:
#include "b.h"
void f(B*);
void f(void*);
void test(D* x) { f(x); }  // calls f(B*)
```

Scoping

Namespaces

Using external libraries should follow a non-declarative convention.

using namespace std; then cout << // Bad use

std::cout<< // Good use

Classes

Structs vs. Classes

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

The struct and class keywords behave almost identically in C++. We add our own semantic meanings to each keyword, so you should use the appropriate keyword for the data-type you're defining.

structs should be used for passive objects that carry data, and may have associated constants. All fields must be public. The struct must not have invariants that imply relationships between different fields, since direct user access to those fields may break those invariants. Constructors, destructors, and helper methods may be present; however, these methods must not require or enforce any invariants.

If more functionality or invariants are required, a class is more appropriate. If in doubt, make it a class.

For consistency with STL, you can use struct instead of class for stateless types, such as traits, template metafunctions, and some functors.

Note that member variables in structs and classes have different naming rules.

Functions

Prefer small and focused functions.

Long functions are sometimes appropriate, so no hard limit is placed on functions length. If a function exceeds about 40 lines, think about whether it can be broken up without harming the structure of the program.

Even if your long function works perfectly now, someone modifying it in a few months may add new behavior. This could result in bugs that are hard to find. Keeping your functions short and simple makes it easier for other people to read and modify your code. Small functions are also easier to test.

You could find long and complicated functions when working with some code. Do not be intimidated by modifying existing code: if working with such a function proves to be difficult, you find that errors are hard to debug, or you want to use a piece of it in several different contexts, consider breaking up the function into smaller and more manageable pieces.

Inputs and Outputs

Input values are provided by parameters. Non-optional input parameters should usually be values or const references.

Output values are provided by return statements only.

Default Arguments

Default arguments are allowed on non-virtual functions when the default is guaranteed to always have the same value. Follow the same restrictions as for function overloading, and prefer overloaded functions if the readability gained with default arguments doesn't outweigh the downsides.

Trailing Return Type Syntax

```
int foo(int x);
```

The newer form, introduced in C++11, uses the `auto` keyword before the function name and a trailing return type after the argument list. For example, the declaration above could equivalently be written:

```
auto foo(int x) -> int;
```

Use the new trailing-return-type form only in cases where it's required (such as lambdas).

Other C++ features

Boost

Use approved libraries only from the Boost library collection.

The Boost library collection is a popular collection of peer-reviewed, free, open-source C++ libraries.

Boost code is generally very high-quality, is widely portable, and fills many important gaps in the C++ standard library, such as type traits and better binders.

Streams

Use streams where appropriate, and stick to "simple" usages. Overload << for streaming only for types representing values, and write only the user-visible value, not any implementation details.

Streams are the standard I/O abstraction in C++, as exemplified by the standard header <iostream>.

The << and >> stream operators provide an API for formatted I/O that is easily learned, portable, reusable, and extensible. printf, by contrast, doesn't even support std::string, to say nothing of user-defined types, and is very difficult to use portably. printf also obliges you to

choose among the numerous slightly different versions of that function, and navigate the dozens of conversion specifiers.

Naming

When following the naming standards in this guide, the most important aspect is consistency. The naming style should vary depending on the entity that is being created: a variable, a function, a constant, a macro, etc. This allows for the name to be quickly associated with its entity type without being necessary to search for its declaration.

Local generated names, i.e., Qt generated names, should keep their original naming style.

General Naming Rules

Names should be directly related to the data/purpose they relate to.

Every entity name should be distinguishable from other names, by referencing the data/purpose they relate to, and maintain good readability.

Abbreviations should only be used for well-known and widely established conventions.

Some names might be fine within smaller scopes, for example "x" might be a valid name within a small 5-line function because its use is very specific and doesn't affect the broader scope. Every name should be proportional to its importance within the broader scope.

File Names

File names should not include any white spaces and be all lowercase. To separate words both "-" and "_" are acceptable but not using symbols is preferred. If two words don't combine well together for readability, prefer using "_".

Examples of acceptable file names:

class_name.cpp -> preferred when word readability is a factor
classname.cpp -> preferred in general
class-name.cpp

Source files should end in .cpp and header files end in .h.

Variable Names

Common Variable Names

Variable names should be mixed case with the first letter in lower case (also known as "camelCase").

Non-Qt generated functions should also follow the "camelCase" naming convention.

Underscores can be used in rare cases where capitalization cannot be used for separation.

Class Data Members

Class variable names follow the same rules as common variables but are prefixed with a "_".

```
std::string     _name;
```

Struct Data Members

Struct variable names follow the same rules as Common Variables.

Constant Names

Constants should be mixed case (also known as "Pascal Case") and prefixed with the letter "k".

const float kPiRounded = 3.14;

Macros

Macros should be all upper case and separated by "_".

#define PI_ROUNDED 3.14

Comments

In general comments directed towards doxygen should follow the doxygen guidelines to generate the reference manual.

All comments not directed towards doxygen should follow the style mentioned below.

Refer to https://www.doxygen.nl/manual/docblocks.html for more information about doxygen comment styles.

Comment style

All **doxygen multi-line comments** should follow the Javadoc style supported by doxygen.

```
/**
 * ... text ...
 */
```

For **doxygen one line comments** use "///".

All **default comments** not directed towards doxygen should use either "//" for one line comments or "/*" for comments with multiple lines.

File comments

For all source and header files, include a small description of the file at the top of each file with author and creation date.

File comments are no to be recognised be doxygen. For this use "/*" to initialise the comment block.

Do not duplicate comments, for example having the same comment in a source file (.cpp) and in the associated header file (.h).

Function comments

Every non-Qt generated function (this means built by the developer and not part of Qt's framework) should have a describing comment at the top where the function purpose, the

parameters and any relevant information should be mentioned. This should be located in the header file and not duplicated in the source file.

Some Qt generated functions can contain comments.

Variable comments

Every variable name should be descriptive enough and not need an associated comment.

For global variables (if unclear) a small comment can be added explained why they are needed.

Formatting

All characters should use utf-8 formatting.

Indentation

Indentation should follow Qt creator's default settings (X spaces when you hit tab).

Curly brackets should be in new lines for multi-lined codeblocks.

Function Declarations and Definitions

Return type on the same line as function name, parameters on the same line if they fit. Wrap parameter lists which do not fit on a single line as you would wrap arguments in a function call.

Functions look like this:

```
ReturnType ClassName::FunctionName(Type par_name1, Type par_name2) {
  DoSomething();
  ...
}
```

If you have too much text to fit on one line:

```
ReturnType ClassName::ReallyLongFunctionName(Type par_name1,
                                             Type par_name2,
                                             Type par_name3) {
  DoSomething();
  ...
}
```

or if you cannot fit even the first parameter:

```
ReturnType LongClassName::ReallyReallyReallyLongFunctionName(
    Type par_name1,
    Type par_name2,
    Type par_name3) {
  DoSomething();
  ...
}
```

Conditionals

In an if statement, including its optional else if and else clauses, put one space between the if and the opening parenthesis, and between the closing parenthesis and the curly brace (if any), but no spaces between the parentheses and the condition or initializer. If the optional initializer is present, put a space or newline after the semicolon, but not before.

Use curly braces for the controlled statements following if, else if and else. Break the line immediately after the opening brace, and immediately before the closing brace. A subsequent else, if any, appears on the same line as the preceding closing brace, separated by a space.

```
if (condition) {                    // no spaces inside parentheses, space
before brace
  DoOneThing();                     // two space indent
  DoAnotherThing();
} else if (int a = f(); a != 3) {   // closing brace on new line, else on
same line
  DoAThirdThing(a);
} else {
  DoNothing();
}
```

Exception Rules

Existing Non-conformant Code

You may diverge from the rules when dealing with code that does not conform to this style guide.

If you find yourself modifying code that was written to specifications other than those presented by this guide, you may have to diverge from these rules in order to stay consistent with the local conventions in that code. If you are in doubt about how to do this, ask the original author or the person currently responsible for the code. Remember that consistency includes local consistency, too.

(Google, n.d.)

References

Google, n.d. *Google Style Guide - C++*. [Online]
Available at: https://google.github.io/styleguide/cppguide.html
[Accessed 8 February 2021].