Coding Standards

v1.0.0

Table of contents

Overview

This is ConstantRobotics' Coding Standards document. This guide outlines essential principles for collaboration, code maintenance, libraries usage, and readability. By following these guidelines, we aim to maintain efficiency, and code quality. As well, this repository provides templates for libraries and applications with documentation example.

Versions

Table 1 - Document versions.

Version	Release date	What's new
1.0.0	04.01.2024	First version of standards, which includes: - Style guide. - Memory management guide. - Templates for library and application with documentation guide. - List of resources.

Coding Guide

The main point of coding guide is to provide a set of main rules that will be fundamental in our repositories structure. Our code should be optimized for the reader, not for the writer. The most important is efficiency, but to build up expandable and easy to maintain coding projects we have to follow basic standards.

General rules

1. Follow the 'Boy Scout Rule'. Always leave the code cleaner than you found it.
2. Everything you write have to be compatible with C++ 17. Any departure from this principle have to be clearly mentioned in repository's documentation.
3. Always include "README.md" file to the repository.
4. If there is a chance that your class will be used as a base class, always keep destructor virtual.
5. No destructor is always better when it's the correct thing to do. If you dont need destructor, dont define the one. This rule is know as Rule of zero.
6. When you define a destructor in a class, it's important to also explicitly define or delete all constructors and assignment operators. This ensures proper management of resources, as the compiler's implicit versions may not handle resource management correctly. This rule is known as Rule of three (until C++11) or Rule of five (after C++11)

Style

Consistency is the most important aspect of style. The second most important aspect is following a style, that the average C++ programmer is used to reading. C++ allows for arbitrary-length identifier names, so there's no reason to be terse when naming things. Use descriptive names and be consistent in the style.

Naming

1. Use **PascalCase** for naming: a class, an enum class or a struct, e.g. - `TemplateLibrary`.
2. Use **camelCase** for naming: a local variable, function or a method, e.g. - `setParam()`.
3. Use **upper case with underscores** for naming: a constant, e.g. - `MAX_CONSTANT_VALUE`.
4. Distinguish private class field with a *'m_'* prefix, e.g. - `m_frameWidth` (`m_` stands for "member").
5. Distinguish global variables with a *'g_'* prefix, e.g. - `g_frameWidth` (`g_` stands for "global").
6. Don't name anything starting with underscore '_'.

Composition

1. In comment blocks use `//`, not `/* */`. `/* */` is reserved for documentation in header file only.
2. Strive to strike a balance in commenting your code - while it is crucial to provide sufficient comments for clarity and understanding, aim to keep them concise and focused, avoiding unnecessary verbosity to maintain code readability and promote efficient collaboration:

```
// Good comment:
// This equation comes from mass-energy equivalence E=mc^2.
auto energy = mass * speedOfLight * speedOfLight;

// Bad comment(code explains itself, comment is redundant):
// Check if energy was calculated.
if (energy.isCalculated()) {}
```

3. Curly brackets `{}` are **required** for blocks. It costs you nothing, but can cost a lot of time for someone changing code after you:

```
// Bad Idea
// This compiles and does what you want, but can lead to confusing
// errors if modification are made in the future and close attention  is not paid.
for (int i = 0; i < 15; ++i)
  std::cout << i << std::endl;

// Good Idea
// It's clear which statements are part of the loop (or if block, or  whatever).
for (int i = 0; i < 15; ++i)
{
  std::cout << i << std::endl;
}
```

4. Take whole line for a single curly bracket (just to keep the same visual style everywhere).

```
// Like this:
if (i == 0)
{
  std::cout << i << std::endl;
}
// Not like this:
if (i == 0) {
  std::cout << i << std::endl;
}
```

5. Keep lines a reasonable length. In header files do not cross length of **80** characters.
6. Namespace name should be descriptive and short. Before adding a new namespace check if it doesn't exist in different form. For example do not add `cr::image` if there is already `cr::frame`.
7. Keep 3-lines space between each method or function body.

Documentation

1. Each class, method, field or function has to have descriptive documentation written as a comment in the header file.
2. Use `/* */` only for documentation:

```
 /**
  * @brief Set the value for a specific template parameter.
  * @param id The identifier of the template parameter.
  * @param value The value to set for the parameter.
  * @return True if the parameter was successfully set, false otherwise.
  */
bool setParam(TemplateParam id, float value);
```

3. Keep the documentation up-to-date when making changes to the code.

Good practices

1. Use `nullptr` to indicate a null pointer. Not `NULL` nor `0`.
2. **Never** use `using namespace` in a header file. Try to avoid also in source C++ files, since it can create name conflicts and ambiguities. Why is `using namespace` is considered a bad practice.
3. Apply `include` guards to avoid including the same file multiple times. Use `#pragma once`.
4. Always use namespaces while creating new class. All namespaces should be placed in one general namespace `cr`.

```
#pragma once

namespace cr::temp
{
class TempClass
{
    TempClass();
};
}
```

5. Always initialize variables. Initialize variables with curly brackets to avoid narrowing chosen value:

```
unsigned value = -1; // Narrowing from signed to unsigned.
unsigned valueB{ -1 }; // Narrowing from signed to unsigned not alllowed, compile
time error.
```

6. When building a class with fields, always initialize member variables in header file, not in the constructor.
7. Use C++-style cast instead of C-style cast. Use the C++-style cast (`static_cast<>, dynamic_cast<>` ...) instead of the C-style cast. The C++-style cast allows more compiler checks and is considerably safer. Additionally the C++ cast style is more visible and has the possibility to search for. Further reading.
8. Avoid using macros for constant values. Use constexpr:
   not `#define PI 3.14159;`,
   but `constexpr double PI = 3.14159;`.

9. Use scoped enums (**enum class**). They prevent implicit conversions to other types, reducing bugs, and avoid name conflicts by scoping enumerators within the enum.

```
enum class Color
{
  RED,
  GREEN,
  BLUE
};
```

10. Use `override` keyword for all polymorphic member functions, this way compiler will report an error if you made mistake while overriding a method.

```
  class Base
  {
    virtual void foo() = 0;
  };

  class Derived : public Base
  {
    void foo() override;
  };
```

11. Try to avoid using global state. If you can't avoid using them at least try to make them internally linked by using unnamed namespaces.

Memory Management

1. Using smart pointers such as `std::unique_ptr` and `std::shared_ptr` in C++11 and later versions is recommended over raw memory access, allocation, and deallocation to mitigate the risks of memory errors and leaks. When using C-style pointers check twice proper memory (de)allocation.
2. Always use `const` if possible, this helps the compiler optimize the code and is more unequivocal for developer.
3. Returning by `&` or `const &` can have significant performance savings when the normal use of the returned value is for observation. Returning by value is better for thread safety and if the normal use of the returned value is to make a copy anyhow, there's no performance lost.
4. Never return `nullptr` from a function. When a function returns pointer and some condition is not satisfied, just return pointer to the empty value.
5. Shared variables. Use `std::atomic` for basic types. Use `std::mutex` library for operating with shared variables, in cases like parameters classes:

```
std::atomic<bool> templateVariable{ false };
templateVariable.store(true);

TemplateParams templateParams{};
std::mutex templateParamsMutex;

templateParamsMutex.lock();
templateParams.firstParam = true;
templateParamsMutex.unlock();
```

6. If semantically correct prefer `++i` over `i++`. Pre-increment is faster than post-increment (it doesn't require copy of the object being made).

Code Management

Commits

1. Commit as often as possible. Don't wait until you finish the task with committing. Wrong combination of `Ctrl+Z` and `Ctrl+Y` may cost you losing a lot of code while prototyping. Frequent commits allow you to make smaller, focused changes to your codebase. Each commit represents a single logical unit of work or a specific feature/fix. This granularity makes it easier to understand the history of changes, identify the cause of issues, and review or revert changes if needed.
2. Write your commit messages in the imperative mood, as it follows same style as commits generated by git commands.

```
# Bad practice. Using dots at the end also bad idea.
git commit -m "Version number in documentation updated."
# Good practice.
git commit -m "Update version number in documentation"
```

3. Make your commit message descriptive, but not longer than 80 chars.

```
# Bad practice.
git commit -m "Fix bug"
# Good practice.
git commit -m "Fix memory leak after copying frame bug"
```

Branches

1. A new Branch should be created each time you want to fix something or add a new feature to the master Branch.
2. Connect Branch with Task from Projects board on GitHub. Create name related to Task name.
3. A Branch should be closed automatically by Pull-request after merge.

Pull-requests

1. Pull-request should be named the same as a branch.
2. Pull-request can be closed only after approval from all reviewers.

Recommended tools and libraries

1. Visual Studio Code - simple and powerful IDE.
2. Visual Studio for Windows - the most complex C++ IDE.
3. Qt Creator - Qt IDE with Qt GUI tools.
4. Sourcetree - very convenient desktop application for git control.

Resources

C++ best practices:

1. https://github.com/cpp-best-practices/cppbestpractices
2. https://google.github.io/styleguide/cppguide.html
3. https://isocpp.org/wiki/faq/coding-standards
4. Clean C++ - Stephan Roth, 2017

Clean Code tips:

1. https://gist.github.com/wojteklu/73c6914cc446146b8b533c0988cf8d29

General standards:

1. https://www.devbridge.com/articles/coding-best-practices/
2. https://medium.com/agile-adapt/composing-meaningful-tasks-c1ca51064c1a