Coding Standards Guide - BFRS

## Summary -

This guide is used to provide the person (tester, programmer, developer) with information as to how the code of this program was written, in order to support future maintenance, in terms of updates and debugging. It is focused on the readability of the program. Each standard used will be provided, along with a description of the chosen coding standard. We at BFRS, have adopted a similar style guide to Google, which can be found here: <https://google.github.io/styleguide/cppguide.html>

## Goals -

The goals of this guide are listed below:

* Optimise code for the reader, not the writer
* Be consistent with existing code
* Be consistent with coding standards of the broader C++ community
* Avoid surprising or dangerous constructs
* Avoid constructs that an average C++ programmer would find hard to maintain
* Concede to optimisation where necessary

## Our Standards -

**Header Files -**

Every cc. file should have an associated .h file. In exception to small .cc files containing just a main() function.

#### Self-contained Headers -

Header files should be self-contained and end in .h. Users should not have to adhere to specific conditions so that they can include the header.

**Comments -**

Comments are incredibly important in order to describe certain parts of code so that they are kept readable. We must also remember that the best code is self-documenting, and should not need too many comments. Comments should be written with an audience in mind, as important as they are for personal use, the next contributor must also understand the code.

#### Comment Style -

Use either the // or the /\* \*/ syntax; however, // is much more common. Be consistent with how you comment and what style you use where.

**Scoping -**

#### Namespaces -

With few exceptions, place code in a namespace. Namespaces should have unique names based on the project name, and possibly its path. Namespaces subdivide the global scope into distinct, named scopes, and so are useful for preventing name collisions in the global scope.

#### Local Variables -

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 data type the variable is and what it was initialised to.

#### Static(Global) Variables -

Use of dynamic initialization for static class member variables or variables at namespace scope is discouraged, but allowed in limited circumstances. They should only be used if objects are trivially indestructible, such as in use for constants, command-line flags, logging, and registration mechanisms.

**Structs and Classes -**

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.

**Functions -**

#### Inputs and Outputs -

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.

**Inclusive Language -**

In all code, including naming and comments, use inclusive language and avoid terms that other programmers might find disrespectful or offensive (such as "master" and "slave", "blacklist" and "whitelist", or "redline"), even if the terms also have an ostensibly neutral meaning. Similarly, use gender-neutral language unless you're referring to a specific person (and using their pronouns). For example, use "they"/"them"/"their" for people of unspecified gender (even when singular), and "it"/"its" for software, computers, and other things that aren't people.

**Naming -**

#### General Naming Rules -

Use names that describe the purpose or intent of the object. Do not worry about saving horizontal space as it is far more important to make your code immediately understandable by a new reader. Minimize the use of abbreviations that would likely be unknown to someone outside your project (especially acronyms and initialisms). Do not abbreviate by deleting letters within a word. As a rule of thumb, an abbreviation is probably OK if it's listed in Wikipedia. Generally speaking, descriptiveness should be proportional to the name's scope of visibility. For example, n may be a fine name within a 5-line function, but within the scope of a class, it's likely too vague.

#### File Names -

Filenames should be all lowercase and can include underscores (\_) or dashes (-). Follow the convention that your project uses. If there is no consistent local pattern to follow, prefer "\_".

#### Type Names -

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 -

The names of variables (including function parameters) and data members are all lowercase, with underscores between words. Data members of classes (but not structs) additionally have trailing underscores.

#### Constant Names -

Variables declared constexpr or const, and whose value is fixed for the duration of the program, are named with a leading "k" followed by mixed case. Underscores can be used as separators in the rare cases where capitalization cannot be used for separation.

#### Function Names -

Regular functions have mixed case; accessors and mutators may be named like variables. Ordinarily, functions should start with a capital letter and have a capital letter for each new word.

#### Namespace Names -

Namespace names are all lower-case, with words separated by underscores. Top-level namespace names are based on the project name. Avoid collisions between nested namespaces and well-known top-level namespaces.

**Formatting -**

Coding style and formatting are often personal preference, 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.

#### Line Length -

Each line of text in the code should be at most 80 characters long. This is excluding include statements, comment lines, text and certain integer outputs, and header guards. When you have a boolean expression that is longer than the standard line length, be consistent in how you break up the lines.

#### Non-ASCII Characters -

Non-ASCII characters should be rare and if used, should use UTF-8 formatting. Hex encoding is also okay, where it enhances readability.

#### Spaces and Tabs -

Use tabs for indentation. Spaces can be used however in our code we have used tabs.