Coding Contribution Standards:

C++ Version:

Please ensure that you use **C++17** throughout your contributions to this project. With regards to IDE preference, for development, our team will use Eclipse 2020-12 as it is currently the most stable for development.

Header Files:

Almost every .cc file should have an associated .h file, except for unit tests and small .cc files containing just a main() function.

Correct use of header files can help improve the readability, size and performance of the code. The following advises on how best to implement header files.

Self-contained Headers

All header files should be self-contained (compile on their own), they should not require specific conditions to be included, and should have header guards and include all other headers it needs. Header files should usually end in .h, with exception of .inc files used for inclusion. .inc files should only be used where a file designed to be included is not self-contained, for example, it may be in an unusual location. They may not use header guards or include their prerequisites. They should see limited use, however, and in all situations, a self-contained header should be prioritised.

Define Guards

Within all header files ensure the use of #Define guards to prevent multiple inclusions and to avoid unnecessary code recursion. Conflicts or recursive errors could result in code failing to build.

Guards should be named uniquely. The standard naming convention is <FILENAME>_H

Include What You Use

The header file should only include all the header files needed for that source and header file, where either uses a symbol defined elsewhere. Transitive inclusions, inclusions where a header is included in one file but both use symbols from each other, should be avoided at all cost. This allows includes to be simply removed without issues being caused elsewhere.

Forward Declarations

Forward declarations, declaration of an entity without an associated definition, should be avoided. While they do improve compile-time and reduce the need for recompilation, they are likely to cause more mistakes and use more lines than just including the header.

Inline Functions

Functions should not be defined inline, with exception of short, performance-critical functions. While inlining of small individual functions may cause them to generate more efficient object code, overuse may cause an overall decline in program speed as the cost is increased. As a rule, functions should not be inline if they are longer than 10 lines.

Names and Order of Includes

To ensure that missing includes are caught early, include headers should be grouped in the following order:

- The Related header e.g. #include "main.h"
- The C system headers e.g. #include <stddef.h>
- The C++ standard library headers e.g. #include <string>
- The other libraries' headers e.g. #include "basictypes.h"
- The project's headers e.g. #include "other.h"

Each group should be separated by one blank line. The exception to this rule is system-specific code which needs conditional includes, these may be put after other includes. System-specific should be small and localized.

Scoping:

Namespaces

Code should be placed under a namespace, which divides the global scope into individual named scopes to prevent name collisions. Namespace names must be unique and should be all lowercase and underscored between words. Avoid using abbreviations unless they follow the rules laid out in the naming section.

Internal Linkage

Sections of code can be given internal linkage using an unnamed namespace (formatted the same as regular namespaces), and individual functions and variables can also be given internal linkage by declaring them as static. Internal linkage prevents whatever has been declared from being accessed from another file. Therefore, internal linkage should be used when the code doesn't need to be referenced elsewhere in .cpp files. It should not be used in `.h` files.

Non-member, Static Member and Global Functions

Always place non-member functions in a namespace and only use global functions when absolutely necessary. Static members should not be grouped. Non-member functions should not depend on external variables and should instead exist in a namespace.

Local Variables

A function variable should be in the narrowest scope possible. Variables should be declared as close as possible to the use. Variables should be initialised when declared, for example; int i = 0. If variables are needed in an object such as an if-statement or for-loop they should be declared just above the constructor.

Static and Global Variables

Objects with static storage duration are disallowed unless there are guaranteed trivial destructors. Global/static variable initialization will depend on the initializer, as a general rule one should always allow for a constant expression. An example of one that is allowed is: int id = getid(); is allowed. Dynamic initialisation of static local variables is permitted.

Classes:

Constructors must not call virtual functions. Do not define an implicit conversion, one should use the explicit keyword for conversion operators and single argument constructors. Type conversion operators should be marked explicit in the class definition. Every class's public interface should say which copy and move operations the class supports which should be done in the public section.

Structs:

Structs should only be used for objects that are passive and carry data otherwise use a class.

Inheritance:

Prioritise composition over inheritance. All inheritance should be public if done privately they try adding it as a member of the base class instead.

Access

Classes' data members should be made private unless they are constants. This helps protect data.

Structure

In general, group similar kinds of declarations together and do not put large method definitions inline in the class definition. See the formatting guide below for more info.

Functions:

Functions should be written using the old-style function definitions for example: string funct(int y); this helps readers and coders who work with other languages understand the code. Default arguments must not be used on virtual functions but they can be used elsewhere.

Short Functions

Functions should always be short and focused; keeping code short helps isolate bugs and testing. Look to break up large functions into smaller ones unless it adds unneeded complexity.

Inputs/Outputs

Use return values when possible over output parameters, this improves readability. Avoid returning pointers unless it is possible for them to be null, preferably return by value, failing that, return by reference. When having non-optional input parameters they should be constant references or values, while output or input/output ones should generally be references.

Style

Cpplint should be used to detect style errors, it is preinstalled on QT creator but can be run separately if needed.

Naming Conventions:

Variables / Functions

Variables will always be named using camel-case. Type names and function names should start with a capital letter for each new word. **DO NOT** use underscores. Class data members should end in an underscore like std::string myvar_;.

Constants / Enumerators

When declaring constants, always capitalise each new word and begin the constant with the letter k, this helps keep code clear. Enumerators should also be named like constants.

Some abbreviations are ok as long as they are common or clear, for example, i/j for iteration or CPU for a central processing unit. When writing a variable think about whether it would hinder the code by using the abbreviation such that it would make it less understandable.

File Names

File names should always be lowercase we will use an underscore '_ 'between words. C++ files should always end in .cpp while header files should end in `.h`. File names should be as specific as possible. Use clear names like:

- main_menu_gui.cpp
- admin_class.cpp
- admin_class_test.cpp

Note: Always end a test file in `_test` for the sake of simplicity.

Comments:

Comments improve readability and make code more accessible to team members and future developers, improving maintainability as well. While commenting is important, good code should be readable without comments; variables and types with good naming should not need a comment to explain them.

Comment Style

The commenting style should be consistent. Comments are to done using the Qt style provided by Doxygen, as seen below:

```
/*! \brief Brief Description
Brief Description continued.


The detailed description starts here.
More detailed description.
*/
```

By using Doxygen, the documentation process can be automated, by developing a Doxyfile.

File Comments

File comments describe the contents of a file. Due to the nature of the project, being non-commercial, the file comments should not need to include an author line or copyright notice. However, if a .h file declares multiple abstractions, a file-level comment should be used to describe the file contents and how the abstractions are related. The comment should be short, as the detailed documentation of individual abstractions should be in the region of said abstractions. If a file declares, implements, or tests exactly one abstraction that is documented by a comment at the point of declaration, then file comments are not required. All other files must have file comments. File comments should not be duplicated in the .cc as well as the .h, as they would likely end up diverging.

Class Comments

Every class or struct declaration where their use is not immediately evident should have a comment that describes its usage. The description should be as clear and concise as possible while covering all relevant information on how to correctly use the class. All information included would extend to how (such as an example use of the class) and when to use the class, and any additional considerations such as synchronisation or threading. Specifically, comments describing the use of a class would be included with its declaration, and comments about the class operation and implementation should accompany the implementation of the class's methods.

Function Comments

Similar to class comments, comments describing the use of a function should be at the function declaration and comments to describe the implementation of the function should be included with the function definition.

Function comments should not be unnecessarily verbose and should only cover what is not immediately obvious. At the function declaration, a comment may be an example of how to use the function. At the function definition, the comments should cover areas where the implementation is overly complex or not clear and not repeat comments in the function declaration. There should be very few comments on the function definition as good code should be clear on what it does without the need for any comments.

Variable Comments

Variables should be named clearly enough to not need comments, however, comments may be needed in certain cases. For class data members, comments may be used for Sentinel Values, those used as flags or pointers, where their use is not immediately clear. Another circumstance is Global Variables, which should cover their use and why they are global.

Implementation Comments

While good code should need few comments, where the implementation may confuse, comments should be used. Complicated blocks of conde may have a proceeding comment to explain the purpose of the entire block. Individual lines of code where their use is not clear, or specific choices may not be clear, may also require commenting to prevent future problems.

Where the arguments of a function are not obvious, comments should be a last resort. Alternatives may include using named constants over literals, avoiding nesting functions in functions over using variables, replacing bool arguments with enum arguments.

Punctuation, Spelling and Grammar

Comments should be easy to read themselves, so they should have good punctuation, spelling, and grammar; including proper capitalisation, punctuation and complete sentences. Shorter, single line, comments may be written in a less formal shorthand as long as they are still readable, however, whole sentences should be the priority.

TO-DO Comments

TODO comments may be used for short-term tasks, such as something you may do the next day. They should be written in the format: // TODO (<NAME>): <TASK>. By using TODO as a standard keyword, TODO messages can be easily searched for using a refactoring tool. TODO messages must be removed when the task is complete. The name attached to the task should almost always be the name of the task writer, to make sure that everyone is aware of who the task belongs to. If the task is addressed to another person, that person should also be informed directly of the task having been added.

Formatting:

Code formatting is arbitrary but following the same guide helps keep consistency and make code easier to understand.

Line length

A line of code should always remain under 80 characters, this helps keep the code readable and understandable. It will be common that function declarations and returns will exceed the 80-character limit in this case it should be broken up onto separate lines.

Parameter names should be short but clear if possible, to assist in keeping code short.

The use of tabs is preferred in this project and should be used in favour of spaces.

Boolean expressions should also be broken up while the logical operators should always be at the end of the line.

Class format

Sections should be placed in order, public, protected and private.

Whitespace

Try to minimise vertical white space, they should be considered the ending to a paragraph and used sparingly to help separate ideas. Overuse of whitespace will make code more difficult to follow. Simply use them where they are thought to be appropriate, such as separating comments and functions.

Other C++ Features:

Boost

Boost Guidelines: https://www.boost.org/doc/libs/1_45_0/libs/test/doc/html/utf/usage-recommendations/generic.html

You should use tests that are specific and precise, try to avoid a complex test in favour of smaller tests. You should prefer `BOOST_CHECK_EQUAL` over `BOOST_CHECK`. This is because you will see the incorrect value when the code failure.

Group tests together

You should aim to group similar tests and name each test so you know what it is checking to help with the debugging later on.