Coding Standards

# C++

## Language Version

All code shall conform to the C++17 standard. Taking full advantage of the features C++17 and the STL has to offer is encouraged, but with consideration to predictable memory use, code performance, and toolchain support.

## Indentation

Tabs shall be used to indent code to the level of the current block, with spaces used after that for readability. This allows each developer to set their preferred tab width while preserving alignment for readability.

Example:

void foo( int param1, int param2

··········int param3, int param4 )

{

-> if ( someLongConditional == someLongFunctionName() &&

-> ·····anotherLongConditional == anotherLongFunctionName() )

-> {

-> -> bar( someLongParam,

-> -> ·····anotherParam );

-> }

}

## Function Overrides

* Non-virtual member functions shall not be overridden in subclasses, as this opens the possibility of different functions being called on the same object for different pointer types.
* The overridespecifier shall be appended to any member functions that override a function from a base class. This will cause the compiler to check that both base function exists and that it is virtual.

## Null Pointers

Pointers shall be set to null using the nullptr keyword.

## Allocation Ownership

Raw pointers should only be used to pass the address of an object with no transfer of ownership, and for storing the address of non-owned objects. std::unique\_ptr shall be used to store pointers which are in the scope of their ownership, and as the return type of functions that pass ownership to the caller. This automates a lot of functionality to prevent memory leaks with near zero performance cost.

std::shared\_ptr and std::weak\_ptr shall be used to store pointers to objects with shared ownership.

## Documentation

Namespaces, classes, member functions, and client accessible functions must be commented for use by Doxygen to generate documentation. Any documentation should be of sufficient detail that examination of implementation should not be necessary to make correct use of a function or class. “JavaDoc” style comment blocks shall be used, and shall be indented to the level of the current block according to indentation standards.

/\*\*

\* Text documenting the class.

\*/

# Source Tree Organization

## Out of Source Builds

Tool chain files and build artifacts shall be located and created outside of the directories containing the source code. This includes, but is not limited to makefiles, IDE solution and project files, CMAKE lists, etc.

## Separation of Headers and Implementation

Client facing headers for a library shall be contained within a folder named according to the library and shall be in separate folders from implementation (\*.cpp) files. Implementation files must reference headers using angle-bracket form (ex. #include <libName/header.h>) since no specific location of client facing headers relative to implementation will be forced in code. Client facing headers shall reference other headers in the same library in quoted form with relative paths (ex. #include "../config.h") and headers in outside libraries using angle-bracket form.

These practices make it very easy for someone who is receiving the source code of a library to integrate it into their project as they see fit.