**Coding Standards and Guidelines**

**1. Coding Standards**

**1.1 Global Variable Usage**

* Strictly limit the use of global variables due to potential side effects and reduced code predictability.
* Clearly define allowed data types for global variable declaration.
* Initialize variables on declaration whenever possible.

**1.2 Header Files**

* Employ header guards to prevent multiple inclusions of the same header file.

#ifndef MY\_HEADER\_H

#define MY\_HEADER\_H

// Header content goes here

#endif // MY\_HEADER\_H

**Naming conventions for header guards:**

* Typically use uppercase letters and underscores.
* Match the filename of the header file (in uppercase). For example, if your header file is named my\_header.h, the guard would be MY\_HEADER\_H.

**1.3 Naming Conventions**

* Utilize meaningful and descriptive names for all variables, functions, classes, and namespaces to enhance code comprehension.
* Adhere to the following conventions:
  + Variables: camelCase for local variables (e.g., localData), PascalCase for member attributes and constants (e.g., MemberVariable, CONSTANT\_VALUE).
  + Functions: camelCase with verbs at the beginning (e.g., calculateArea).
  + Struct: PascalCase (e.g., MyClass) should be nouns.
  + Namespaces: camelCase separated by double colons (e.g., myProject::utilityFunctions).

**1.4 Indentation**

* Maintain consistent indentation (typically 4 spaces) for improved code readability.
* Indent nested code blocks to visually represent their hierarchy.

**1.5 Braces**

* Always use braces around code blocks, even for single-line statements, to improve readability and maintainability.

**1.6 Error Handling**

* Establish a standardized error handling approach. Consider using exceptions or error codes with clear and descriptive error messages.

**1.7 Pointers and References**

* Employ pointers and references judiciously, and document their usage clearly to avoid memory management issues and dangling pointers.
* Utilize smart pointers (e.g., std::unique\_ptr, std::shared\_ptr) for automatic memory management when appropriate.

**2. Coding Guidelines**

**2.1 Meaningful Comments**

* Avoid single-line comments that simply restate the code.
* Use comments to explain the "why" behind code decisions, not just the "what."

**2.2 Function Length**

* Strive for functions to be less than 50 lines of code (excluding comments) for better readability and maintainability.
* Decompose large functions into smaller, more manageable units with clear responsibilities.

**3. Additional Considerations**

**3.1 Build System and Code Formatting**

* Employ a build system (e.g., Make, CMake) to automate the compilation and linking process.
* Utilize code formatting tools (e.g., Clang-Format, AStyle) to enforce consistent coding style across the project.

**3.2 Unit Testing**

* Implement unit testing practices using frameworks like Google Test or Catch2 to ensure the functionality of individual code units.

**3.4 Reference Resources**

[Google C++ Style Guide](https://students.cs.byu.edu/~cs235ta/references/Cpp%20Style/Google%20Cpp%20Style%20Guide.pdf)