Contents

SIMDIS SDK Code Style Guide

From Section 2.1.5 of DISA Application Security and Development STIG

Implementing coding standards provides many benefits to the development process. These benefits include readability, consistency, and ease of integration.

Code conforming to a standard format is easier to read, especially if someone other than the original developer is examining the code. In addition, formatted code can be debugged and corrected faster than unformatted code.

Introducing coding standards can help increase the consistency, reliability, and security of the application by ensuring common programming structures and tasks are handled by similar methods, as well as, reducing the occurrence of common logic errors.

Coding standards also allow developers to quickly adapt to code which has been developed by various members of a development team. Coding standards are useful in the code review process as well as in situations where a team member leaves and duties must then be assigned to another team member. Coding standards often cover the use of white space characters, variable naming conventions, function naming conventions, and comment styles.

Definitions

The word *shall* will be used to indicate a mandatory requirement, one that must be followed. Deviation from a rule of this type requires a detailed justification to be placed in comments within the code and/or the documentation for the code.

The word *should* will be used to indicate important suggestions that a developer ought to strongly consider when writing code. These kinds of rules will not be enforced but suggestions may be made at code reviews for reconsideration. No comments or documentation are necessary to deviate from one of these rules.

The terminology being used within this document is modeled after the suggested standards documentation rules as specified by the IEEE. The rules herein are paraphrases of section 13.1 at http://standards.ieee.org/guides/style/section5.html, customized to be more applicable to SIMDIS SDK development.

Files

- Auto-generated files, such as those from `protoc`, are largely excluded from the coding style guidelines.
- All header and source files *shall* include the boilerplate header template shown below.

  Toggle line numbers

```
/* -*- mode: c++ -*- */
/*****************************************************************************
 *****                                                                   *****
 *****                   Classification: UNCLASSIFIED                    *****
 *****                    Classified By:                                 *****
 *****                    Declassify On:                                 *****
 *****                                                                   *****
 *****************************************************************************
 *
 *
 * Developed by: Naval Research Laboratory, Tactical Electronic Warfare Div.
 *               EW Modeling & Simulation, Code 5773
 *               4555 Overlook Ave.
 *               Washington, D.C. 20375-5339
 *
 * License for source code can be found at:
 * https://github.com/USNavalResearchLaboratory/simdissdk/blob/master/LICENSE.txt
 *
 * The U.S. Government retains all rights to use, duplicate, distribute,
 * disclose, or release this software.
 *
 */
```

- Header files *shall* be named with the `.h` extension. Source files *shall* be named with the `.cpp` extension. Inline header files may be used, and *shall* be named with a `-inl.h` extension.
    - Good: `Platform.h`, `TextFormatter.cpp`, `MemoryDataSlice-inl.h`
    - Bad: `Platform.cc`, `Registry.hpp`, `AnimatedLine.inl`, `Interpolator.hh`, `GenericIterator.hxx`
- File names *shall* start with a capital letter and have a capital letter for each new word, without underscores or spaces.

  - Good: `Platform.h`, `TextFormatter.cpp`, `MemoryDataSlice-inl.h`
  - Bad: `platformModel.h`, `Linear Interpolator.h`, `Table_Status.h`
- Source files *should* be named for the class defined in the file. Additional helper classes may also be defined in the header.
  - Good: `simData/DataStore.h` and `simData/DataStore.cpp` include definitions and implementation for the `simData::DataStore` class.
- Other source files, such as conglomerates of related classes, convenience functions, compatibility headers, etc., *should* have names reflect of their use.
  - Good: `simCore/Time/String.h` defines time string functions, including a base class `TimeFormatter`, several derived implementations, and a `TimeFormatterRegistry` for composite operations.

Naming and Identifiers

- Variable names *should* have names reflective of their role in the problem domain.
  - Good: `xInertial`, `shipBodyCoordinates`
  - Bad: `foo`, `tmp`, `x`
- Variable names *should not* contain articles like `the`, `a`, `an`, etc. and *should* be of appropriate length for their scope.
  - Good: `coordinateSystem`, `dataStore`, `x` (when used as a looping control variable)
  - Bad: `aCoordinate`, `theDataStore`, `x` (outside of a loop control value)
- Abbreviations *should* be avoided, especially in class and file names. Commonly recognized abbreviations are permitted. Abbreviations and acronyms are subjected to the coding style guidelines and *should* only capitalize the first letter in an abbreviation or acronym. Loop control values like iterators can be exempt.
  - Good: `class PlatformDialog`, `UrlManager.h`, `SdkVersion.cpp`, `class MemoryDataStore`
  - Bad: `class PlatformDlg`, `URLManager.h`, `SDKVersion.cpp`, `class MemDS`
- Types (including classes, enumerations, and typedefs) *shall* be written in Upper Camel Case, also sometimes known as Pascal Case, e.g. `UpperCamelCase`, where the initial letter of each word in the identifier is capitalized, including the first word.
  - Good: `enum Choices`, `typedef std::vector<int> IntVector`
  - Bad: `enum CHOICES`, `enum values`, `class helperClass`, `typedef std::vector<int> int_vector`
- Any identifier created *shall* not have the suffix _t as that is reserved by the POSIX standard
  - Good: `enum Choices`, `typedef std::vector<int> IntVector`
  - Bad: `enum Choices_t`, `typedef std::vector<int> IntVector_t`
- Variables and functions, such as class methods, global functions, local variables, etc. *shall* be written in Lower Camel Case where the initial letter of each word in the identifier is capitalized, not including the first word.
  - Good: `int calculate()`, `double calculateValue()`, `IntVector intVector;`
  - Bad: `int Calculate()`, `double calculatevalue()`, `double calculate_value()`, `IntVector Values`, `float DISTANCE`
- For both types and instances, acronyms *shall* be treated as lowercase for the purpose of inclusion in the identifier name.
  - Good: `class HtmlReader`, `int readHtml(const std::string& htmlData)`
  - Bad: `class HTMLReader`, `int readHTML(const std::string& htmlData)`
- Accessor functions for class members *should* be named such that the setter has the 'set' prefix and the getter has no prefix.
  - Good:

    Toggle line numbers

    ```
    class ValueCache
    {
    public:
      int value() const
      {
        return value_;
      }
      void setValue(int value)
      {
        value_ = value;
      }
    private:
      int value_;
    };
    ```

  - Bad:

    Toggle line numbers

    ```
    class ValueCache
    {
    public:
      int getValue() const
      {
        return value_;
      }
      void value(int value)
      {
        value_ = value;
      }
    private:
      int value_;
    };
    ```

- Preprocessor macros *shall* be written in all upper case with underscores as word separator, e.g. MACRO_NAME(), emphasizing their unusual scoping, i.e. none.
  - Good: `SIM_ERROR`, `STDOUT_FILENO`

  - Bad: `SimError`, `STDOUT_FileNo`
- Enumeration values *shall* be written in all upper case with underscores as word separator, like a preprocessor substitution.
  - Good: `enum DataTypes { INTEGER, DOUBLE, BOOLEAN }`
  - Bad: `enum DataTypes { Integer, Double, boolean }`
- Exceptions *should* have the suffix Exception appended to their name to indicate their intended use.
  - Good: `TimeException`, `TableException`
  - Bad: `InvalidTime`, `NotEnoughRows`
- Functions receiving no parameters *shall* use empty parentheses in their signatures, and not `(void)`.
  - Good: `int calculate() const`
  - Bad: `int calculate(void) const`
- Private and protected class member variables and methods *shall* have the suffix _ appended to their names.
  - Good:

    Toggle line numbers

    ```
    class ValueCache
    {
    public:
      int value() const
      {
        return value_;
      }
      void setValue(int value)
      {
        value_ = value;
      }
    private:
      void clearValue_()
      {
        value_ = 0;
      }
      int value_;
    };
    ```

  - Bad:

    Toggle line numbers

    ```
    class ValueCache
    {
    public:
      int value() const
      {
        return _value;
      }
      void setValue(int value)
      {
        _value = value;
      }
    private:
      void clearValue()
      {
        value_ = 0;
      }
      int _value;
    };
    ```

- The `NULL` keyword *should* be preferred over `0L` and `nullptr`. `nullptr` *shall* only be used when `NULL` creates ambiguity and/or compiler errors.
  - Good: `platformNode = NULL;`
  - Good: `std::vector<ColumnCellPair>::const_iterator i = std::lower_bound(vec.begin(), vec.end(), ColumnCellPair(columnI`
    - Note that `nullptr` is required in this statement on some supported compilers
  - Bad: `platformNode = 0L;`
  - Bad: `platformNode = nullptr;`

Scoping Rules

- Namespaces *shall* be preferred to prefixes when attempting to avoid name clashes.
  - Good: `FX::Class`, `simData::DataStore`
  - Bad: `FXClass`, `SimDataDataStore`
- Static methods of classes *shall* be referenced using the class scope operator (`::`) both when called from within a class and without, and *shall not* include spaces on either side of the `::` operator.
  - Good: `OrdinalTimeFormatter::toStream(...)`, `simCore::StringUtils::before("ie", "Good")`
  - Bad: `OrdinalTimeFormatter format; format.toStream(...)`, `this->staticMethod(...)`
- Preprocessor macros, due to their unusual scoping (i.e. none), *shall* be replaced by `static const` variables where appropriate and reserved only for when they are required to accomplish a lexical substitution.
  - Good: `static const double PI = 3.14159;`, `#define SIM_ERROR SIM_NOTIFY(simCore::NOTIFY_ERROR)`
  - Bad: `#define PI 3.14159`, `#define SQRT(x) (x*x)`
- The `using` directive *shall* only be used in header files within a class or function definition and not at file scope.

  - Good: `using namespace std` in `simCore/EM/RadarCrossSection.cpp`
  - Bad: `using namespace std` in `simCore/EM/RadarCrossSection.h`
- The comma operator *shall not* be used unless extremely localized and documented as to its purpose. The comma operator *shall not* be used in the 1st or 3rd clause of a `for` statement; effort can be made to rewrite such for loops as while statements.
  - Good: `for (int k = 0; k < 5; ++k)`
  - Bad: `return x = 5, 2 * x;`, `for (int k = 0; k < 5; ++k, j += 2)`
- Header files *shall* contain include guards that are unique to each file to prevent multiple inclusion within a compilation unit. The macro chosen *should* be the filename, possibly with important namespace directory information, uppercased with an `_H` appended to prevent conflicts with other identifiers.
  - Good: For `simData/DataStore.h`:

    Toggle line numbers

    ```
    #ifndef SIMDATA_DATASTORE_H
    #define SIMDATA_DATASTORE_H
    ...
    #endif /* SIMDATA_DATSTORE_H */

    ```

  - Bad: For `simData/MemoryDataStore.h`:

    Toggle line numbers

    ```
    #ifndef _MEMORY_DATA_STORE_H_
    #define _MEMORY_DATA_STORE_H_
    ...
    #endif /* _MEMORY_DATA_STORE_H_ */

    ```

- Classes *shall* have their protection scopes written as `public` before `protected` before `private`, because the `public` interface is most important to those developers reading only a header file.
  - Good:

    Toggle line numbers

    ```
    class Example
    {
    public:
      int calculate();
    protected:
      int preCalculate_();
    private:
      int cachedCalculation_() const;
    };
    ```

  - Bad:

    Toggle line numbers

    ```
    class BadExample : public QObject
    {
      Q_OBJECT;
    private:
      int cachedCalculation_() const;
    private slots:
      int recalculate_();
    public:
      int calculate();
    protected:
      int preCalculate_();
    };
    ```

- Local header files *shall* be `#included` using `""` to surround the included filename while standard header files that do not change very often *shall* be `#included` using `<>` to surround the included filename.
  - Good: `#include <cassert>`, `#include "simData/DataStore.h"`
  - Bad: `#include "cassert"`, `#include <simData/DataStore.h>`
- Header files *should* be included with full path prefix information for clarity.
  - Good: `#include "simData/DataStore.h"` from a file in the `simData` directory
  - Bad: `#include "DataStore.h"` from a file in the `simData` directory
- Header files that are `#included` *shall* be ordered as system header files at the top, external dependency header files in the middle, and local header files at the bottom of the included file list.
  - Good:

    Toggle line numbers

    ```
    #include <cassert>
    #include <QObject>
    #include "simQt/TimeButtons.h"

    ```

  - Bad:

    Toggle line numbers

    ```
    #include "simQt/TimeButtons.h"
    #include <QObject>
    ```

```
#include <cassert>

```

- All code *should* be place in a namespace. Source code *should* reside in a directory named for the associated namespace and the associated `#include` *should* reflect this directory structure.
  - Good: `#include "simData/DataStore.h"` for class `DataStore` in namespace `simData`.
  - Bad: Not using a namespace or subdirectory.
- Static class member functions *should* be used in place of nonmember functions or global functions. Global functions *shall* not be used.
  - Good:

    Toggle line numbers

    ```
    class StringUtils
    {
    public:
      static std::string before(const std::string& in, const std::string& str);
    ```

  - Bad:

    Toggle line numbers

    ```
    static std::string before(const std::string& in, const std::string& str);
    ```

- Local variables *shall* be defined as close to use as possible, and *shall* be initialized in the declaration.
  - Good:

    Toggle line numbers

    ```
    int value = 5;
    int result = calculate_(value);
    ```

  - Bad:

    Toggle line numbers

    ```
    int value;
    int result;
    value = 10;
    result = calculate_(value);
    ```

- All non-`const` class data member variables *shall* be declared `private`. Derived classes *shall* use `protected` or `public` methods to manipulate `private` variables as necessary, and *shall not* have direct access to inherited variables.
  - Good:

    Toggle line numbers

    ```
    class EncapsulatedValue
    {
    public:
      static const int INVALID_VALUE = 0;
      static const float RADIANS_TO_DEGREES;
    private:
      double width_;
    };
    ```

  - Bad:

    Toggle line numbers

    ```
    class NotEncapsulatedValue
    {
    public:
      double width;
    };
    ```

- Global variables (including globals in namespaces) *shall not* be used. Singleton pattern *should not* be used in general, and *shall not* be used in any situation where threading might be used. Global constants in an appropriate namespace are encouraged.

Declarations

- Size-specific types (such as `uint32_t` or `int8_t`) *should not* be used unless the data must be read from a file or network and must match a particular memory layout; however, even this deviation *should* be documented.
  - Good:

    Toggle line numbers

    ```
    double width = 0;
    int secondsSinceMidnight = 20;
    std::vector<uint8_t> byteArray;
    ```

  - Bad:

    Toggle line numbers

    ```
    int32_t secondsSinceMidnight = 0;
    std::vector<uint16_t> favoriteUnsignedIntegers;
    ```

- Magic numbers *shall* be avoided except when initializing well-documented constants or variables.
  - Good: `static const double PI = 3.14159;`
  - Bad: `double angleDegrees = angleRadians * (180/3.14159);`
- Arguments *should* be passed by reference if `NULL` values are not possible. An object *shall* be passed as `const T&` if its value will not be modified. An object *shall* be passed as `T&` if its value may be modified.
  - Good: `bool isValidNumber(const std::string& token, double& val, bool permitPlusToken=true);`
  - Bad: `bool isValidNumber(std::string token, double* value);`
- The `assert` macro *shall* be used only to verify preconditions, postconditions and invariants that are always true regardless of input or user behavior and *shall* include no code that has side effects. The `assert` *shall* be documented with reasoning and steps to take if assertion is triggered.
  - Good:

    Toggle line numbers

    ```
    // Previous logic prevents negative angle; if fails, check logic above.
    assert(angle >= 0.0);
    ```

  - Bad:

    Toggle line numbers

    ```
    assert(angle % 360 > 100);  // NB: No clue how to fix this?
    assert(isValid = (angle >= 0));  // NB: Side effect

    ```

- Result codes *shall* be checked to ensure that no error conditions have been triggered. Errors *shall not* be ignored.
  - Good:

    Toggle line numbers

    ```
    if (mightFail_() != 0)
      return 1;
    return 0;
    ```

  - Bad:

    Toggle line numbers

    ```
    mightFail_();
    return 0;
    ```

- Integer result code *shall* be preferred; `0` *shall* be used for valid return values (only one way to succeed), and non-zero *shall* be used for invalid return values. Complex error returns *should* be encoded into an enumeration. Boolean values *shall not* be used as a generic error code.
  - Good:

    Toggle line numbers

    ```
    int checkRange(int value);
    int calculateSqrt(double input, double& output); // Returns non-zero on error (negative input)
    bool isValidValue(int value);
    TableStatus addRow();  // Complex enumeration of possible failures

    ```

  - Bad:

    Toggle line numbers

    ```
    bool checkRange(int value);  // What does true mean?  What does false mean?
    bool calculateSqrt(double input, double& output);
    ```

- Deeply nested code blocks *shall* be avoided by reimplementing the algorithm or refactoring into or more functions. Functions longer than 30 lines *should* be avoided.
  - Good:

    Toggle line numbers

    ```
    void testColor(int row, int col)
    {
      for (int colorIndex = 0; colorIndex < 4; ++colorIndex)
      {
        if (bytes[row].getValue(col, colorIndex))
          ...
      }
    }
    void searchArray(...)
    {
      for (int row = 0; row < height; ++row)
      {
        for (int col = 0; col < width; ++col)
        {
          testValue(row, col);
        }
      }
    }
    ```

  - Bad:

    Toggle line numbers

```
void searchArray(...)
{
  for (int row = 0; row < height; ++row)
  {
    for (int col = 0; col < width; ++col)
    {
      for (int colorIndex = 0; colorIndex < 4; ++colorIndex)
      {
        if (bytes[row].getValue(col, colorIndex))
          ...
```

- Parentheses *shall* be used to impose the order of operations within complicated expressions, unless the only operators within the expression are the well-understood `+`, `-`, `*` or `/`.
  - Good: `return value + (flags << 2);`
  - Bad: `return value + flags << 2;`
- All assignment statements *shall* be on a line by themselves except in `for` or `while` loop statements.
  - Good:

    Toggle line numbers

    ```
    int value1 = 0;
    int value2 = 2;
    value0 = 4;
    for (int k = 5; k < 100; ++k) ...
    ```

  - Bad:

    Toggle line numbers

    ```
    int value1 = 0, value2 = 2;
    value0 = 4, value2 = 5;
    ```

- Casts *shall* be C++ style using `static_cast<>`, `dynamic_cast<>`, and `reinterpret_cast<>`, as needed.
  - Good: `int value = static_cast<int>(angle);`, `simVis::PlatformNode* platform = dynamic_cast<simVis::PlatformNode*>(node);`
  - Bad: `int value = (int)angle;`, `simVis::PlatformNode* platform = (simVis::Platform*)node;`
- All variables *should* be initialized at declaration or documented as to why the initialization is unnecessary.
  - Good: `int lineLength; // initialized by simCore::isValidNumber() below`
  - Bad: `int width;`
- All class data members *shall* be initialized explicitly in the member initialization list or within the constructor body and in the order that they are declared within the body of the class. Constructors in source (`.cpp`) files *shall* contain no more than one variable initialization per line, and commas *shall* be placed at the end of the line.
  - Good:

    Toggle line numbers

    ```
    class Example
    {
    public:
      Example() : one_(1), two_(2) {}
    private:
      int one_;
      int two_;
    };
    ```

  - Good:

    Toggle line numbers

    ```
    Example::Example()
     : one_(1),
       two_(2)
    {
    }
    ```

  - Bad:

    Toggle line numbers

    ```
    class Example
    {
    public:
      Example() : three_(3), one_(1) {} // Oops, two_ is not initialized, and three_ initialized out of order
    private:
      int one_;
      int two_;
      int three_;
    };
    ```

  - Bad:

    Toggle line numbers

    ```
    Example::Example() : one_(1), two_(2)
    {
    }
    ```

    - Bad:

      Toggle line numbers

      ```
      Example::Example()
       : one_(1)
       , two_(2)
      {
      }
      ```

- Constructors that take one argument *should* be declared `explicit` if the class cannot be implicitly constructed from that type in a valid manner.
    - Good:

      Toggle line numbers

      ```
      class RangeTool
      {
        explicit RangeTool(ScenarioManager* scenario);
      ```

    - Bad:

      Toggle line numbers

      ```
      class RangeTool
      {
        RangeTool(ScenarioManager* scenario);
      ```

Indentation and Spacing

- Unless specified otherwise in this document, all spacing and indentation style *shall* follow the Allman style, also known as ANSI style.
- Code body indentation *shall* be two spaces per line of code. Tabs *should not* be used in lieu of spaces. Tabs *shall* represent eight spaces.
    - Good:

      Toggle line numbers

      ```
      int noop()
      {
        return 0;
      }
      ```

- All opening and closing brackets *should* be placed on their own lines. Namespace declarations are exempt.
    - Good:

      Toggle line numbers

      ```
      namespace simData { class ForwardDeclaration; }
      namespace simUtil {
      class Example
      {
      public:
        simData::ForwardDeclaration* instance();
      };
      }
      ```

    - Bad:

      Toggle line numbers

      ```
      namespace simData { class ForwardDeclaration1;
        class ForwardDeclaration2;
      }
      class Example {
      public:
        simData::ForwardDeclaration2* instance() { return NULL; }
      };
      ```

      Toggle line numbers

      ```
      if (isValid())
      {
        executeStage1_();
      }
      else { return 1; }
      ```

- Spaces *shall not* be used after a function name and before the opening parenthesis.
    - Good: `setValue(100);`, `return doFunction();`, `for (int k = 0; k < test(); ++k)`
    - Bad: `setValue (100);`, `return doFunction ();`, `for(int k = 0; k < test(); ++k)`
- A single space *shall* be used after each comma in an argument list.
    - Good: `return doFunction(var1, var2, var3);`
    - Bad: `return doFunction(var1,   var2,var3);`
- A single space *shall* follow C++ reserved words before opening parentheses.
    - Good: `if (x)`, `while (notDone())`
    - Bad: `if(x)`, `while(notDone())`
- Keywords `private`, `protected`, and `public` *shall* be left-aligned with the `class` keyword.

  - Good:

    Toggle line numbers

    ```
    class GoodAlignment
    {
    public:
      GoodAlignment();
    };
    ```

  - Bad:

    Toggle line numbers

    ```
    class BadAlignment
    {
      public:
        BadAlignment();
    };
    ```

- Braces *should not* be added to `case` statements in a switch unless variable declaration is required inside the case. When braces are used in a `case` statement, the `break` statement *shall* be placed inside the braces.
  - Good:

    Toggle line numbers

    ```
    switch(var)
    {
      case 0:
        return 1;
      case 1:
      {
        int test = sqrt(var);
        break;
      }
    }
    ```

  - Bad:

    Toggle line numbers

    ```
    switch(var)
    {
      case 0:
      {
        return 1;
      }
      case 1:
      {
        int test = sqrt(var);
        break;
      }
    }
    ```

- Falling through a `case` statement by omitting a `break` statement *shall* be documented when intentional
  - Good:

    Toggle line numbers

    ```
    switch (var)
    {
      case ERROR1:
      case ERROR2: // No documentation necessary
        std::cout << "Error encountered." << std::endl;
        // Fall-through is intentional
      case NO_ERROR:
        break;
    }
    ```

Comments and Doxygen Formatting

- Public classes, methods, functions, and macros *shall* be documented with an appropriate Doxygen documentation block. Nested classes *shall* be treated the same as public classes for the purposes of documentation. Long-style Doxygen comments *should* be preferred, providing appropriate `@param` and `@return` statements.
- Public class and global variables, constants, typedefs, enumerated values, and other identifiers *should* be documented with an appropriate Doxygen documentation block.
- All source code *shall* have an appropriate amount of internal documentation. Code *should* be documented with the understanding that others will be reading, reviewing, and modifying the code you write over the next dozen or more years.
- Doxygen level documentation *shall* focus on intent ("why" over "what") and/or context, and not be a simple rewording of the method or variable name. Helpful items to include in documentation are purpose, examples, caveats, special considerations, and anything else that might not be immediately obvious by looking at the function signature.
  - Good:

    Toggle line numbers

    ```
    /** Maintains consistent index in platform list for centering properly as platform list changes */
    class CenterHelper
    ```

- Bad:

  Toggle line numbers

  ```
  /** A class to help with centering */
  class CenterHelper
  ```

- Documentation blocks *shall* be placed in header files, associated with declarations. Documentation blocks *shall not* be duplicated in both the header declaration and the source code implementation.
- If any method parameter or return value is documented explicitly using the `@param` or `@return` directive, then all of the method's parameters and return value *shall* be documented explicitly. A documentation block *should not* consist of only `@param` or `@return` directives. `@param` directives *shall* be in order, based on the order of parameters to the method, with `@return` (if needed) last.
  - Good:

    Toggle line numbers

    ```
    /**
     * Performs mathematical exponent operation of "base" to the "power" power.
     * @param base Base of the exponentiation operation ("base" to the "power" power, or base^power)
     * @param power Exponent or power to apply to the base in exponentiation
     * @return Value of base to the power (base^power) using mathematical exponentiation.
     */
    double exponent(double base, double power) const;
    ```

  - Acceptable:

    Toggle line numbers

    ```
    /** Performs mathematical exponent operation of "base" to the "power" power. */
    double exponent(double base, double power) const;
    ```

  - Bad:

    Toggle line numbers

    ```
    /**
     * Performs mathematical exponent operation of "base" to the "power" power.
     * @param base Base of the exponentiation operation ("base" to the "power" power, or base^power)
     */
    double exponent(double base, double power) const;
    ```

  - Bad:

    Toggle line numbers

    ```
    /**
     * Performs mathematical exponent operation of "base" to the "power" power.
     * @param base Base of the exponentiation operation ("base" to the "power" power, or base^power)
     * @param power
     * @return Value of base to the power (base^power) using mathematical exponentiation.
     */
    double exponent(double base, double power) const;
    ```

  - Bad:

    Toggle line numbers

    ```
    /** @return Mathematical exponent operation of "base" to the "power" power. */
    double exponent(double base, double power) const;
    ```

  - Bad:

    Toggle line numbers

    ```
    /**
     * Performs mathematical exponent operation of "base" to the "power" power.
     * @param power Exponent or power to apply to the base in exponentiation
     * @param base Base of the exponentiation operation ("base" to the "power" power, or base^power)
     * @return Value of base to the power (base^power) using mathematical exponentiation.
     */
    double exponent(double base, double power) const;
    ```

- Triple slash notation *shall* only be used for single line "short" documentation without `@param` or `@return` statements.
  - Good:

    Toggle line numbers

    ```
    /// Returns true when the position has been set and time is within a valid epoch.
    bool isValid() const;
    ```

  - Good:

    Toggle line numbers

    ```
    /** Returns true when the position has been set and time is within a valid epoch. */
    bool isValid() const;
    ```

  - Good:

    Toggle line numbers

```
/**
 * Detects validity and usability of the current instance based on position and time.
 * @return True when position has been set and time is within a valid epoch.
 */
bool isValid() const;
```

  - Bad:

    Toggle line numbers

    ```
    /// @return true when the position has been set and time is within a valid epoch
    bool isValid() const;
    ```

  - Bad:

    Toggle line numbers

    ```
    /// Detects validity and usability of the current instance based on position and time.
    /// @return True when position has been set and time is within a valid epoch.
    bool isValid() const;
    ```

  - Bad:

    Toggle line numbers

    ```
    /// Detects validity and usability of the current instance
    /// based on position and time.
    bool isValid() const;
    ```

- Parameter names *shall* follow `@param` directive, and not the data type. Use of the `[in]`, `[out]`, and `[in,out]` *should* be used when the tags help to clarify intent, and are to be placed between the `@param` and variable name.
  - Good: `@param base Base of the exponentiation operation ("base" to the "power" power, or base^power)`
  - Good: `@param[in] base Base of the exponentiation operation ("base" to the "power" power, or base^power)`
  - Good: `@param[out] units Parsed units value from the input string.`
  - Good: `@param[in,out] scaledValue Double scalar value in meters, to be scaled to the specified units on return.`
  - Bad: `@param double base Base of the exponentiation operation ("base" to the "power" power, or base^power)`
- Long-style comments *shall* either be a single line, or the starting (`/**`) and ending (`*/`) tokens *should* be on lines to themselves.
  - Good:

    Toggle line numbers

    ```
    /** Detects validity and usability of the current instance based on position and time. */
    bool isValid() const;
    ```

  - Good:

    Toggle line numbers

    ```
    /**
     * Detects validity and usability of the current instance
     * based on position and time.
     */
    bool isValid() const;
    ```

  - Bad:

    Toggle line numbers

    ```
    /** Detects validity and usability of the current instance
     *  based on position and time.
     */
    bool isValid() const;
    ```

  - Bad:

    Toggle line numbers

    ```
    /**
     * Detects validity and usability of the current instance
     * based on position and time. */
    bool isValid() const;
    ```

- Comment lines on multiline long-style comments *shall* start with a * and space token prior to other text.
  - Good:

    Toggle line numbers

    ```
    /**
     * Detects validity and usability of the current instance
     * based on position and time.
     */
    ```

  - Bad:

    Toggle line numbers

```
/**
  Detects validity and usability of the current instance
  based on position and time.
 */
```

- Bad:

  Toggle line numbers

  ```
  /**
   *Detects validity and usability of the current instance
   *based on position and time.
   */
  ```

- Documentation blocks *should* be placed before the code being documented. Avoid the `<` notation.
  - Good:

    Toggle line numbers

    ```
    /// List of all top level views (not maintained directly by ViewManager)
    QList<ViewObserverPtr> topLevelViews_;
    ```

  - Bad:

    Toggle line numbers

    ```
    QList<ViewObserverPtr> topLevelViews_;  ///< List of all top level views (not maintained directly by ViewManager)

    ```

- You may use the `@copydoc` syntax to reuse documentation from a parent or inherited method. For example:

  ```
  class SDKVIS_EXPORT ResolvedPositionOrientationLocator : public Locator
  {
    // ...
    /** @copydoc Locator::getPosition_() */
    virtual bool getPosition_(osg::Matrixd& pos, unsigned int comps) const;
  ```

Unsafe Functions, Input Validation, and Overflows

Secure coding practices *shall* be used to prevent attackers from exploiting developed software to compromise the integrity and reliability of a system. Good rules to follow when writing safe code can be found in Bjarne Stroustrup's C++ FAQ (http://www2.research.att.com/~bs/bs_faq.html#unsafe):

> Why does C++ support operations that can be used to violate the rules of static (compile-time) type safety?
> - to access hardware directly (e.g. to treat an integer as a pointer to (address of) a device register)
> - to achieve optimal run-time and space performance (e.g. unchecked access to elements of an array and unchecked access to an object through a pointer)
> - to be compatible with C
>
> That said, it is a good idea to avoid unsafe code whenever you don't actually need one of those three features:
> - don't use casts
> - keep arrays out of interfaces (hide them in the innards of high-performance functions and classes where they are needed and write the rest of the program using proper strings, vectors, etc.)
> - avoid `void*` (keep them inside low-level functions and data structures if you really need them and present type safe interfaces, usually templates, to your users)
> - avoid `union`
> - if you have any doubts about the validity of a pointer, use a smart pointer instead,
> - don't use "naked" news and deletes (use containers, resource handles, etc., instead)
> - don't use `...`-style variadic functions ("`printf` style")
> - Avoid macros except for include guards
>
> Almost all C++ code can follow these simple rules. Please don't be confused by the fact that you cannot follow these rules if you write C code or C-style code in C++.

The most important issues to avoid through the use of secure coding practices are buffer overflow (or underwrite) vulnerabilities exposed by the use of unsafe functions, generally functions which do not check array bounds, and format string attacks. More information about buffer overflow, buffer underwrite, and format string attacks can be found at:

- Buffer overflow vulnerability: http://www.owasp.org/index.php/Buffer_Overflow
- Buffer underwrite vulnerability: http://www.owasp.org/index.php/Buffer_underwrite
- Format string attack: http://www.owasp.org/index.php/Format_string_attack

The following rules are intended to help prevent overflow and format string attacks by restricting the use of unsafe functions which make them possible.

- The following is a list of unsafe functions that *shall not* be used.

Unsafe Functions	
Function	**Alternative**
`strcpy`	Standard Template Library string class and operations *should* be used whenever possible. If required to work with C-style strings, use `strncpy` instead, but be aware that `strncpy` only limits the number of bytes copied and does not check memory boundaries.
`strcat`	Standard Template Library string class and operations *should* be used whenever possible. If required to work with C-style strings

	use, `strncat` instead, but be aware that `strncat` only limits the number of bytes copied and does not check memory boundaries.
`gets`	C++ `iostream` operations *should* be used for reading strings from stdin whenever possible. If required to work with C functions, use `fgets` instead.
`printf`	The use of `printf` with the `%n` format identifier *shall not* be allowed.
`sprintf`	C++ `stringstream` operations or alternatively the Boost Format library *should* be used for formatting strings. If required to work with C functions, use `snprintf` instead.
`vsprintf`	C++ `stringstream` operations *should* be used for formatting strings. If required to work with C functions, use `vsnprintf` instead.

- Automated checks *shall* be used to identify the presence of unsafe functions alert developers to their presence. For example, in Microsoft Visual C++, warning `C4996` indicates an unsafe function call. Use of static code analysis *shall* be used to identify reported unsafe function calls. Appendix B of the DISA Application Security and Development STIG v3r1 identifies functions that have a greater potential to cause application vulnerabilities. It is noted that the presence of these functions does not indicate a vulnerability; however the way they are used may cause a vulnerability.
- Standard Template Library (STL) string classes *should* be used to replace documented unsafe functions and character arrays to minimize buffer overflow vulnerabilities.
- STL stream operations *should* be used instead of the printf family of functions to minimize format string vulnerabilities.
- The use of compile-time options that add compiler buffer overrun defenses *shall* be used.
  - MSVC: `-GS` and `-EHa`
  - g++: `-fstack-protector`
- The use of compile-time options that detect insecure use of format strings at runtime *shall* be used.
  - g++: `-Wformat-security`
- The `size_t` type *shall* be used to minimize integer overflows when performing pointer arithmetic, array indexing, and specifying array size and allocation, in order to minimize integer overflows.
- Mixing `signed` and `unsigned` data types, as well as data types of different sizes, *should* be avoided. The use of compiler warnings and static code analysis *shall* be used to detect these problems.
- Input validation *shall* be used before passing data to an interpreter or compiler in order to minimize command injection vulnerabilities.
- In multi-threaded applications the use of global and static variables *shall* be minimized. The use of thread-safe and re-entrant versions of functions *shall* be used. Applications for checking and verifying multi-threaded code, such as Intel Thread Checker *should* be used.

Special Circumstances

- Consistency is important to readability. Consistency with existing style (pre-existing non-conformant code) *should* take precedence over following the style dictated in this guide. However, efforts *should* be made to fix style violations as feasible.
- All `osg::Referenced`-derived classes *shall* have a protected destructor.
- All `osg::Referenced`-derived class instances held in class member variables *shall* be contained in an `osg::ref_ptr` or `osg::observer_ptr` to clarify pointer ownership.
- All code *shall* compile correctly on all supported platforms, both in static and dynamic (`dll` or `so`) CMake configurations. This means appropriate use of `SDKCORE_EXPORT`, `SDKDATA_EXPORT`, etc., and minimizing cross-platform differences in code. Current compilers include Microsoft Visual Studio 9.0, 10.0, 11.0, and 12.0 in 32 and 64 bit configurations, and g++ 4.1 through 4.6.
- Smart pointers (`std::tr1::shared_ptr<TypeName>`) *should* be used to convey shared ownership when necessary, particularly when implementing the Observer pattern. `osg::Referenced`-derived classes *shall not* be placed into a `shared_ptr`, but use the `osg::ref_ptr` instead.
- External or new internal dependencies *shall not* be permitted in the base SIMDIS SDK modules (`simCore`, `simData`, `simVis`, `simUtil`, and `simQt`) without approval from the configuration control board.
- The `simQt` module and all Qt example code *shall* configure and build without error against Qt 4.8, and Qt 5.2 (or newer).
- Warnings in SIMDIS SDK code *shall not* be permitted, except as a direct and unavoidable use of third party headers (such as warnings in Protobuf header files).