Team Ember C++ Development Guidelines

Jason Lefley and Richard Greene

09/03/2015

# Introduction

This document provides guidelines for C++ development. Consideration of the provided guidelines helps achieve stylistic consistency and reliable, extensible, and maintainable code, while avoiding common pitfalls. This guide is heavily based on [1] and [2].

# General Design

Note that the following guidelines will not guarantee the production of an error-free product. However, adherence to these guidelines, in addition to any process guidelines, will help programmers produce clean designs that minimize common sources of mistakes and errors.

In general, the code produced should exhibit the following important qualities:

1. Reliability: Executable code should consistently fulfill all requirements in a predictable manner.
2. Maintainability: Source code should be written in a manner that is consistent, readable, simple in design, and easy to debug.
3. Testability: Source code should be written to facilitate testability. Minimizing the following characteristics for each software module will facilitate a more testable and maintainable module:
   1. code size
   2. complexity
   3. static path count (number of paths through a piece of code)
4. Reusability: The design of reusable components is encouraged. Component reuse can eliminate redundant development and test activities (i.e. reduce costs).
5. Extensibility: Requirements are expected to evolve over the life of a product. Thus, a system should be developed in an extensible manner (i.e. perturbations in requirements may be managed through local extensions rather than wholesale modifications).
6. Readability: Source code should be written in a manner that is easy to read, understand and comprehend.

## Coupling and Cohesion

Coupling and cohesion are properties of a system that has been decomposed into modules.

Cohesion is a measure of how well the parts in the same module fit together. Coupling is a measure of the amount of interaction between the different modules in a system. Thus, cohesion deals with the elements within a module (how well-suited elements are to be part of the same module) while coupling deals with the relationships among modules (how tightly modules are glued together).  
  
Object-oriented design and implementation generally support desirable coupling and cohesion characteristics. The design principles behind OO techniques lead to data cohesion within modules. Clean interfaces between modules enable the modules to be loosely coupled. Moreover, data encapsulation and data protection mechanisms provide a means to help enforce the coupling and cohesion goals.

Source code should be developed as a set of modules as loosely coupled as is reasonably feasible. Note that generic programming (which requires the use of templates) allows source code to be written with loose coupling and without runtime overhead.

Examples of tightly coupled software would include the following:

* many functions tied closely to hardware or other external software sources, and
* many functions accessing global data.

There may be times where tightly coupled software is unavoidable, but its use should be both minimized and localized as suggested by the following guidelines:

* limit hardware and external software interfaces to a small number of functions,
* minimize the use of global data, and
* minimize the exposure of implementation details.

# C++ Coding Standards

The purpose of the following recommendations is to define a style for programs written in C++ that will enable programmers to produce code that is more correct, reliable, and maintainable. In order to achieve these goals, programs should have a consistent style, be free of common types of errors, and be understandable, and hence maintainable by different programmers. These guidelines are not strict rules and meeting the aforementioned goals may warrant deviations. Discuss significant departures from these guidelines with your team before making the proposed changes. Note: Adherence to these guidelines is not strictly required when writing test code. However, keep these guidelines in mind and strive to achieve the same goals as for production code when writing tests.

## Pre-Processing Directives

Since the pre-processor knows nothing about C++, it should not be used to do what can otherwise be done in C++.

1. Only use the following pre-processor directives:
   1. #ifndef
   2. #define
   3. #endif
   4. #include
2. Use #ifndef, #define and #endif to prevent multiple inclusions of the same header file. Do not use other techniques to prevent the multiple inclusions of header files.  
   Rationale: Eliminate multiple inclusions of the same header file in a standard way.  
   Example:  
   For SomeHeaderFileName.h  
   #ifndef SOMEHEADERFILENAME\_H   
   #define SOMEHEADERFILENAME\_H  
    // Header declarations...  
   #endif // SOMEHEADERFILENAME\_H  
   Note: In the case of a nested directory structure of header files, begin the include guard symbol definition name with the directory name(s) (relative to a common root) and replace the slash in the path with an underscore “\_” (i.e. DIR1\_DIR2\_HEADER\_H for dir1/dir2/header.h). This prevents naming conflicts when multiple header files in the same project share the same filename but reside in different directories.
3. Use #ifndef and #endif pre-processor directives only as defined in the previous guideline to prevent multiple inclusions of the same header file.  
   Rationale: Conditional code compilation should be kept to a minimum as it can significantly obscure testing and maintenance efforts.
4. Do not use the #define pre-processor directive to create inline macros. Use inline functions instead.  
   Rationale: Inline functions do not require text substitutions and behave well when called with arguments (e.g. type checking is performed). See AV Rule 29 in Appendix A of [1] for an example.   
   Exception: Some libraries require definition of a specific macro to function properly or as a means of customization. Under these circumstances, defining an inline macro is allowed.
5. Do not use the #define pre-processor directive to define constant values. Instead, apply the const qualifier to variable declarations to specify constant values.  
   Exception: The only exception to this rule is for constants that are commonly defined by third-party modules. For example, #define is typically used to define NULL in standard header files. Consequently, NULL may be treated as a macro for compatibility with third-party tools.  
   Rationale: const variables follow scope rules, are subject to type checking and do not require text substitutions (which can be confusing or misleading). See AV Rule 30 in Appendix A of [1] for an example.
6. Use the #define pre-processor directive only as part of the technique to prevent multiple inclusions of the same header file.
7. Use the #include pre-processor directive only to include header (\*.h) files.  
   Exception: In the case of template class or function definitions, the code may be partitioned into separate header and implementation files. In this case, the implementation file may be included as a part of the header file. The implementation file is logically a part of the header and is not separately compilable.  
   Rationale: Clarity. The only files included in a .cpp file should be the relevant header (\*.h) files.

## Header Files

1. Use the <filename.h> notation to include header files installed at a system-level and the “filename.h” notation to include header files included as part of the project when using the #include directive.  
   Rationale: The “filename.h” notation gives priority to the include directories specified as arguments to the compiler. Leveraging this search order resolves naming conflicts between system and project header files.
2. In a given implementation file (i.e. foo.cpp), order #include statements as follows:
   1. the header file corresponding to the implementation file (i.e. foo.h)
   2. C and C++ system files
   3. other .h files from the project

Rationale: If foo.h omits any necessary #include statements, the build of foo.cpp will break. This ensures that build breaks appear for the people working on foo.h and foo.cpp rather than people using foo.h elsewhere.

1. Compose header files to contain logically related declarations only.  
   Rationale: Minimize unnecessary dependencies.
2. Use a mechanism to prevent multiple inclusions of a header file.  
   Rationale: Avoid accidental header file recursion. Note the Pre-Processor Directives guidelines specify the mechanism that eliminates multiple inclusions whereas this guideline specifies that each header file must use that mechanism.
3. Minimize compilation dependencies when possible.  
   Rationale: Minimize unnecessary recompilation of source files. See AV Rule 36 in Appendix A of [1] for an example.
4. Include only those header files required for successful compilation of a given header (include) in it. Include files that are only used by the associated .cpp file in the .cpp file – not the .h file.  
   Rationale: The #include statements in a header file define the dependencies of the file. Fewer dependencies imply looser couplings and hence a smaller ripple effect when the header file is required to change.
5. Supply declarations of classes that are only accessed via pointers (\*) or references (&) with forward declarations or forward headers that contain only forward declarations.  
   Rationale: The header files of classes that are only referenced via pointers or references need not be included. Doing so often increases the coupling between classes, leading to increased compilation dependencies as well as greater maintenance efforts. Forward declarations of the classes in question (possibly supplied by forward headers) can be used to limit implementation dependencies, maintenance efforts and compile times. See AV Rule 38 in Appendix A of [1] for an example. Note that this technique is employed in the standard header <iosfwd> to declare forward references to template classes used throughout <iostreams>.
6. Do not place non-constant variable definitions or function definitions in header (\*.h) files.  
   Rationale: Header files should typically contain interface declarations – not implementation details.  
   Exception: Inline functions and template definitions may be included in header files. See AV Rule 39 in Appendix A of [1] for an example.

## Implementation Files

1. Include the header files that uniquely define the inline functions, types, and templates used in every implementation file.  
   Rationale: Insures consistency checks.  
   Note that this rule implies that the definition of a particular inline function, type, or template will never occur in multiple header files.

## Style

Imposing constraints on the format of syntactic elements makes source code easier to read due to consistency in form and appearance. Note that automatic code generators should be configured to produce code that conforms to the style guidelines where possible.

1. Use uppercase rather than lowercase letters in literal suffixes.  
   Rationale: Readability.  
   Example:  
   const int64 frameRate = 64l; // Bad, looks too much like 641  
   const int64 frameRate = 64L; // Good
2. Keep source lines to a length of 80 characters or less.  
   Rationale: Readability and style. Very long source lines can be difficult to read and understand.  
   Exception: If a word boundary falls on the 80th column, exceeding 80 characters on that line is acceptable if doing so improves readability. String constants are also permitted to exceed the 80-character limit.
3. Each expression-statement will be on a separate line.  
   Rationale: Simplicity, readability, and style. See AV Rule 42 in Appendix A of [1] for examples.
4. Do not use tabs.  
   Rationale: Tabs are interpreted differently across various editors and printers.
5. Make all indentations 4 spaces.  
   Rationale: Readability and style.
6. Format and indent constructor initialization lists in the following manner:  
   SomeClass::SomeClass() :  
   \_memberA(123),  
   \_memberB(“abc”)  
   {  
    …  
   }

### Naming Identifiers

The choice of identifier names should:

* Suggest the usage of the identifier.
* Consist of a descriptive name that is short yet meaningful.
* Be long enough to avoid name conflicts, but not excessive in length.
* Include abbreviations that are generally accepted.

Note: In general, the above guidelines should be followed. However, conventional usage of simple identifiers (i, x, y, p, etc.) in small scopes can lead to cleaner code and will therefore are permitted.

Additionally, the term ‘word’ in the following naming convention rules may be used to refer to a word, an acronym, an abbreviation, or a number.

1. Separate words in an identifier using camel case.  
   Exception: Separate words in constant identifiers with “\_”.
2. Do not create multiple identifiers that differ by:
   1. Only a mixture of case
   2. The presence/absence of the underscore character
   3. The interchange of the letter ‘O’, with the number ‘0’ or the letter ‘D’
   4. The interchange of the letter ‘I’, with the number ‘1’ or the letter ‘l’
   5. The interchange of the letter ‘S’ with the number ‘5’
   6. The interchange of the letter ‘Z’ with the number 2
   7. The interchange of the letter ‘n’ with the letter ‘h’

Rationale: Readability.

1. Compose all acronyms in an identifier using uppercase letters.  
   Note: Always make an acronym uppercase, even if the acronym is located in a portion of an identifier that is specified to be lower case by other rules.  
   Rationale: Readability.
2. Begin the first word of the name of a class, structure, namespace, enumeration, or type created with typedef with an uppercase letter.  
   Example:  
   class DiagonalMatrix { … };  
   enum RGBColors { Red, Green, Blue };  
   Exception: The first letter of a typedef name may be in lowercase in order to conform to a standard library interface or when used as a replacement for fundamental types.
3. Begin all function names with upper case letters.
4. Begin all non-constant variable names and function parameters with lower case letters.
5. Begin all enumerator values with upper case letters.
6. Begin all class data members with “\_” (underscore).  
   Rationale: Prevent ambiguity when a locally scoped variable has the same name as a class member variable.  
   Note: Do not prefix struct data members with underscores. Since structs are restricted to simple data aggregation, the possibility for naming within the struct is eliminated. Additionally, since structs have public members, prefixing with underscores clutters the external API of the struct.
7. Prefix accessor methods with Get (e.g. GetData()) and mutator methods with Set (e.g. SetData(int data))
8. Make constant variable names entirely uppercase and separated with “\_”.  
   Example:  
   const int MAX\_PRESSURE = 100;
9. Use “.h” as the file name extension for header files.
10. Use “.cpp” as the file name extension for implementation files.
11. Name header files to reflect the logical entity for which it provides declarations.  
    Example: For the Matrix entity, name the header file Matrix.h
12. Name implementation files to reflect the logical entity for which it provides definitions and have a “.cpp” extension (this name will normally be identical to the header file that provides the corresponding declarations.)  
    At times, more than one .cpp file for a given logical entity will be required. In these cases, a suffix should be appended to reflect a logical differentiation.  
    Example 1: One .cpp file for the Matrix class:  
    Matrix.cpp  
    Example 2: Multiple files for a math library:  
    Math\_sqrt.cpp  
    Math\_sin.cpp  
    Math\_cos.cpp

### Classes

1. Declare the public, protected, and private sections of a class in that order (the public section is declared before the protected section which is declared before the private section).  
   Rationale: By placing the public section first, everything that is of interest to a user is gathered in the beginning of the class definition. The protected section may be of interest to designers when considering inheriting from the class. The private section contains details that should be of the least general interest.
2. Add the Interface suffix to names of abstract base classes containing only pure virtual functions that define interfaces.  
   Rationale: Make the distinction between interfaces and implementations explicit.
3. Indent the public, protected, and private visibility modifiers at the same level as the class name declaration itself.  
   Example:  
   class MyClass  
   {  
   public:  
    MyClass();  
    ~MyClass();  
    …  
   }  
   Rationale: Eliminate unnecessary indentation.

### Functions

1. Place the return type, function name, and parameters on the same line if they fit. If they do not fit, prefer to start the argument(s) on a new line after an indentation.

Rationale: Readability.  
Example:  
ReturnType SomeFunction(Type arg1, Type arg2);

ReturnType SomeLongFunction(  
 Type arg1, Type arg2,

Type arg3, Type arg4);  
Exception: Placing more than one indentation or additional spaces before arguments placed on new lines is permitted if doing so improves readability.

1. Wrap function calls that do not fit on a single line in the same fashion as function declarations and definitions (specified in previous guideline).  
   Rationale: Readability.
2. Do not needlessly surround return expressions with parentheses. Use parentheses in return expr; only where you would use them in x = expr;.

Example:

return (result); // Bad  
return result; // Good  
return (someCondition && otherCondition) // Good

### Blocks

1. Place braces ("{}") that enclose a block in the same column, on separate lines directly before and after the block.  
   Example:  
   if (var\_name == true)  
   {  
   }  
   else  
   {  
   }
2. Do not place anything else on a line containing the braces ("{}") that enclose a block except comments (if necessary).

### Conditionals

1. Place a space between the if/else if and the following opening parenthesis.  
   Rationale: Readability.  
   Example:  
   if(condition) // Bad  
   if (condition) // Good

### Pointers and References

1. Directly connect the dereference operator ‘\*’ and the address-of operator ‘&’ with the type-specifier.  
   Rationale: The int32\* p; form emphasizes type over syntax while the int32 \*p; form emphasizes syntax over type. Although both forms are equally valid C++, the heavy emphasis on types in C++ suggests that int32\* p; is the preferable form.  
   Examples:  
   int32\* p; // Preferred  
   int32 \*p; // Not preferred
2. Prefix the names of pointer variables with ‘p’.  
   Rationale: Clarity.

### Miscellaneous

1. Do not use spaces around ‘.’ or ‘->’, nor between unary operators and operands.  
   Rationale: Readability and style.
2. Do not put spaces inside parentheses.  
   Rationale: Style.  
   Example:  
   if ( condition ) // Bad  
   if (condition) // Good

## Classes

1. Create complete and minimal class interfaces.  
   Rationale: A complete interface allows clients to do anything they may reasonably want to do. On the other hand, a minimal interface will contain as few functions as possible (i.e. no two functions will provide overlapping services). Hence, the interface will be no more complicated than it has to be while allowing clients to perform whatever activities are reasonable for them to expect.  
   Note: Overlapping services may be required where efficiency requirements dictate. Also, the use of helper functions can simplify class interfaces.

### Considerations

Roughly two types of classes exist: those that essentially aggregate data and those that provide an abstraction while maintaining a well defined state or invariant (e.g. a property that holds for all instances of a class). Invariants are established during construction and constantly maintained between calls to public methods. The following rules provide guidance in this regard.

1. Use a structure to aggregate publically accessible data (i.e. an entity that does not require an invariant).
2. Use a class to model an entity containing private data that is operated on through its member functions (i.e. an entity that maintains an invariant).
3. Use public and protected data in structs and private data in classes.  
   Rationale: A class is able to maintain its invariant by controlling access to its data. However, a class cannot control access to its members if those members non-private. Hence all data in a class should be private.  
   Exception: Protected members may be used in a class as long as that class does not participate in a client interface.

### Member Functions

1. Explicitly disallow unneeded implicitly generated member functions.  
   Rationale: Eliminate any surprises that may occur as a result of compiler-generated functions. For example, if the assignment operator is unneeded for a particular class, then it should be declared private (and not defined). Any attempt to invoke the operator will result in a compile-time error. On the contrary, if the assignment operator is not declared, then when it is invoked, a compiler-generated form will be created and subsequently executed. This could lead to unexpected results.  
   Note: If the copy constructor is explicitly disallowed, the assignment operator should be as well.

### const Member Functions

1. Always declare a member function that does not affect the state of an object (its instance variables) as const.  
   Member functions should be const by default. Only when there is a clear, explicit reason should the const modifier on member functions be omitted.  
   Rationale: Declaring a member function const is a means of ensuring that objects will not be modified when they should not. Furthermore, C++ allows member functions to be overloaded on their const-ness.

### Friends

1. Add friends to a class only when a function or object requires access to the private elements of the class, but is unable to be a member of the class for logical or efficiency reasons.  
   Rationale: The overuse of friends leads to code that is both difficult to understand and maintain.  
   AV Rule 70 in Appendix A of [1] provides examples of acceptable uses of friends. Note that the alternative to friendship in some instances is to expose more internal detail than is necessary. In those cases friendship is not only allowed, but is the preferable option.

### Constructors

1. Do not invoke a class’s virtual functions from its destructor or any of its constructors.  
   Rationale: A class’s virtual functions are resolved statically (not dynamically) in its constructors and destructor. See AV Rule 71.1 in Appendix A of [1] for additional details.
2. Do not define unnecessary default constructors.  
   Rationale: Discourage programmers from creating objects until the requisite data is available for complete object construction (i.e. prevent objects from being created in a partially initialized state). See AV Rule 73 in Appendix A of [1] for examples.
3. Initialize all member variables such that an object is in a valid and useable state as soon as it is constructed. Use in-class initializers, initialization lists, or assignment in the constructor body.  
   Rationale: Avoid using uninitialized variables.
4. Perform initialization of non-static class members through the member initialization list rather than through assignment in the body of a constructor.  
   Exception: Use assignment when an initial value cannot be represented by a simple expression (e.g. initialization of array values), or when a name must be introduced before it can be initialized (e.g. value received via an input stream). See AV Rule 74 in Appendix A of [1] for details.
5. List members of the initialization list in the order in which they are declared in the class.  
   Note: Since base class members are initialized before derived class members, base class initializers should appear at the beginning of the member initialization list.  
   Rationale: Members of a class are initialized in the order in which they are declared – not the order in which they appear in the initialization list. To avoid falsely relying on initialization list order as the order of initialization, order initialization lists so they match the variable declaration order.
6. Declare a copy constructor and an assignment operator for classes that contain pointers to data items or nontrivial destructors.  
   Rationale: Ensure resources are appropriately managed during copy and assignment operations. See AV Rule 76 in Appendix A of [1] for additional details.
7. Explicitly disallow copy constructors and assignment operators for classes that contain references as data.  
   Rationale: References in C++ cannot be assigned and thus it is nearly impossible to properly define a copy constructor or assignment operator for classes containing references.
8. Define a copy constructor such that it copies all data members and bases that affect the class invariant (a data element representing a cache, for example, would not need to be copied).  
   Note: If a class employs a reference counting mechanism, a literal copy need not be performed in every case.  
   Rationale: Ensure data members and bases are properly handled when an object is copied. See AV Rule 77 in Appendix A of [1] for additional details.
9. Do not define a constructor having default arguments that produce a signature identical to that of the implicitly declared copy constructor for the corresponding class/structure.  
   Rationale: Compilers are not required to diagnose this ambiguity. See AV Rule 77.1 in Appendix A for additional details.

### Destructors

1. Define a virtual destructor for all base classes with a virtual function.  
   Rationale: Prevent undefined behavior. If an application attempts to delete a derived class object through a base class pointer, the result is undefined if the base class’s destructor is non-virtual.  
   Note: This rule does not imply the use of dynamic memory (allocation / de-allocation from the free store) will be used.
2. Release all resources acquired by a class in the class’s destructor.  
   Rationale: Prevention of resource leaks, especially in error cases.

### Assignment Operators

1. Use the default copy and assignment operators for classes when those operators offer reasonable semantics.  
   Rationale: The default versions are more likely to be correct, easier to maintain and efficient than that generated by hand.
2. Handle self-assignment correctly in an assignment operator.  
   Rationale: a = a; must function correctly. See AV Rule 81 in Appendix A of [1] for examples.
3. Return a reference to \*this from an assignment operator.  
   Rationale: Both the standard library types and the built-in types behave in this manner.
4. Assign all data members and bases that affect the class invariant (a data element representing a cache, for example, would not need to be copied) in an assignment operator.  
   Note: To correctly copy a stateful virtual base in a portable manner, it must hold that if x1 and x2 are objects of virtual base X, then x1=x2; x1=x2; must be semantically equivalent to x1=x2;  
   Rationale: Ensure data members and bases are properly handled under assignment. See AV Rule 83 in Appendix A of [1] for additional details.

### Operator Overloading

1. Use operator overloading sparingly and in a conventional manner.  
   Rationale: Since unconventional or inconsistent uses of operator overloading can easily lead to confusion, operator overloads should only be used to enhance clarity and should follow the natural meanings and conventions of the language. For instance, a C++ operator "+=" shall have the same meaning as "+" and "=".
2. When two operators are opposites (such as == and !=), define both and define one in terms of the other.  
   Rationale: If operator==() is supplied, then one could reasonable expect that operator!=() would be supplied as well. Furthermore, defining one in terms of the other simplifies maintenance.

### Inheritance

1. Use concrete types to represent simple, independent concepts.  
   Rationale: Well-designed concrete classes tend to be efficient in both space and time, have minimal dependencies on other classes, and tend to be both comprehensible and usable in isolation.
2. Use abstract base classes containing only pure virtual functions to represent interfaces exposed by multiple implementations. Avoid state and constructors in interfaces.  
   Rationale: Explicit interfaces distinguish how objects communicate from their implementations. Programming against well-defined interfaces insulates clients from implementation changes within their collaborators. Focusing on how objects communicate rather than what they do exposes insights into the system and can help to evolve towards a good design.
3. Base hierarchies on abstract classes.  
   Hierarchies based on abstract classes tend to focus designs toward producing clean interfaces, keep implementation details out of interfaces, and minimize compilation dependencies while allowing alternative implementations to coexist. See AV Rule 87 in Appendix A of [1] for examples.
4. Use multiple inheritance only for implementing multiple interfaces within a single class.  
   Rationale: Multiple inheritance can lead to complicated inheritance hierarchies that are difficult to comprehend and maintain.
5. Do not make a base class both virtual and non-virtual in the same hierarchy.  
   Rationale: Hierarchy becomes difficult to comprehend and use.
6. Make heavily used interfaces minimal, general and abstract  
   Rationale: Enable interfaces to exhibit stability in the face of changes to their hierarchies.
7. Prefer composition to implementation inheritance (public or private). Consider a design revision (e.g. add an instance of the intended base class as a member of the intended derived class) if implementation inheritance seems necessary.  
   Rationale: Implementation inheritance couples a base type’s implementation to that of a derived type. Changes to the implementation of the base type can break functionality of all derived types.
8. Do not redefine an inherited non-virtual function in a derived class.  
   Rationale: Prevent an object from exhibiting “two-faced” behavior. See AV Rule 94 in Appendix A of [1] for an example.
9. Do not redefine an inherited default parameter.  
   Rationale: The redefinition of default parameters for virtual functions often produces surprising results. Se AV Rule 95 in Appendix A of [1] for an example.
10. Do not treat arrays polymorphically.  
    Rationale: Array indexing in C/C++ is implemented as pointer arithmetic. Hence, a[i] is equivalent to a + i \* SIZEOF(array element). Since derived classes are often larger than base classes, polymorphism and pointer arithmetic are not compatible techniques.
11. Do not use arrays in interfaces. Instead, use a different type to wrap the array data.  
    Rationale: Arrays degenerate to pointers when passed as parameters. This “array decay” problem has long been known to be a source of errors.

### Virtual Member Functions

1. Do not make either operand of an equality operator (== or !=) a pointer to a virtual member function.  
   Rationale: If either operand of an equality operator (== or !=) is a pointer to a virtual member function, the result is unspecified

## Namespaces

1. Use namespaces to avoid name clashes in large programs with many parts.
2. Do not nest namespaces more than two levels deep.  
   Rationale: Simplicity and clarity. Deeply nested namespaces can be difficult to comprehend and use correctly.
3. Avoid using directives. Prefer using declarations or explicit qualification.  
   Note: Do not use using namespace std. Bringing in the entire std namespace can cause non-obvious ambiguities.  
   Rationale: All elements in a namespace need not be pulled in if only a few elements are actually needed.

## Templates

1. Avoid complicated templates and prefer class inheritance. The following scenarios may warrant the use of templates:
   1. Run-time efficiency is at a premium.
   2. No common base type can be defined.
   3. Built-in types and structures with compatibility constraints are important.

Rationale: Templates can make code more difficult to understand and require care to ensure proper use. See section 4.12 of [1] for more information.

## Functions

### Function Declaration, Definition and Arguments

1. Always declare functions at file scope.  
   Rationale: Declaring functions at block scope may be confusing.
2. Do not use functions with variable numbers of arguments.  
   Note: In some cases, default arguments and overloading are alternatives to the variable arguments feature. See AV Rule 108 in Appendix A of [1] for an example.  
   Rationale: The variable argument feature is difficult to use in a type-safe manner (i.e. typical language checking rules aren’t applied to the additional parameters).
3. Do not place a function definition in a class specification unless the function is intended to be inlined.  
   Rationale: Class specifications are less compact and more difficult to read when they include implementations of member functions. Consequently, it is often preferable to place member function implementations in a separate file. However, including the implementation in the specification instructs the compiler to inline the method (if possible). Since inlining short functions can save both time and space, functions intended to be inlined may appear in the class specification. See AV Rule 109 in Appendix A of [1] for an example.
4. Consider refactoring or leveraging additional abstractions if a function requires more than three or four parameters.  
   Rationale: Functions having long argument lists can be difficult to read, use, and maintain. Functions with too many parameters may indicate an under use of objects and abstractions.
5. Do not return a pointer or reference to a non-static local object from a function.  
   Rationale: After return, the local object will no longer exist.
6. Do not obscure resource ownership through function return values.  
   Note: Use smart pointers when returning a pointer to heap allocated memory.  
   Rationale: Potential source of resource leaks. See AV Rule 173 and AV Rule 112 in Appendix A of [1] for examples.
7. Write small and focused functions and methods.  
   Rationale: Keeping your functions short and simple makes it easier for other people to read and modify your code.

### Return Types and Values

1. Use return statements for all exit points of value-returning functions.  
   Rationale: Flowing off the end of a value-returning function results in undefined behavior.
2. Prefer to return results using return rather than output parameters.  
   Rationale: Clarifies distinction between function inputs and output. Compiler-provided return value optimization reduces the overhead otherwise incurred by copying a return value.

### Function Parameters (Value, Pointer or Reference)

1. Pass small, concrete-type arguments (two or three words in size) by value if changes made to formal parameters should not be reflected in the calling function.  
   Rationale: Pass-by-value is the simplest, safest method for small objects of concrete type. Note that non-concrete objects must be passed by pointer or reference to realize polymorphic behavior.
2. Pass arguments by reference if null values are not possible:
   1. An object should be passed as const T& if the function need not change the value of the object.
   2. An object should be passed as T& if the function may change the value of the object.

Rationale: Since references cannot be null, checks for null values will be eliminated from the code. Furthermore, references offer a more convenient notation than pointers.

1. Pass arguments via pointers if null values are possible:
   1. An object should be passed as const T\* if its value need not be modified.
   2. An object should be passed as T\* if its value may be modified.

Rationale: References cannot be null.

### Function Overloading

1. Use overloaded operations or methods to form families that use the same semantics, share the same name, have the same purpose, and that are differentiated by formal parameters.  
   Rationale: Inconsistent use of overloading can lead to considerable confusion. See AV Rule 120 in Appendix A of [1] for examples.

### Inline Functions

1. Follow C++ language rules regarding inlining functions:
   1. An inline function shall be defined in every translation unit in which it is used and shall have exactly the same definition in every case. (Note this observation implies that inline function definitions should be included in header files.)
   2. If a function with external linkage is declared inline in one translation unit, it shall be declared inline in all translation units in which it appears; no diagnostic is required.
   3. An inline function with external linkage shall have the same address in all translation units.
   4. A static local variable in an extern inline function always refers to the same object.
   5. A string literal in an extern inline function is the same object in different translation units.
2. Avoid inlining functions with more than two statements.  
   Rationale: Long inlined functions can make class declarations more difficult to read. Additionally, compilers may be more likely to ignore requests to inline longer functions.
3. Inline trivial accessor and mutator functions.  
   Rationale: Inlining short, simple functions can save both time and space.
4. Minimize the number of accessor and mutator functions.  
   Rationale: Numerous accessors and mutators may indicate that a class simply serves to aggregate a collection of data rather than to embody an abstraction with a well-defined state or invariant. In this case, a struct with public data may be a better alternative.
5. Inline trivial forwarding functions.  
   Rationale: Inlining short, simple functions can save both time and space.

### Temporary Objects

1. Avoid unnecessary temporary objects.  
   Rationale: Since the creation and destruction of temporary objects that are either large or involve complicated constructions can result in significant performance penalties, they should be avoided. See AV Rule 125 in Appendix A of [1] for additional details.

## Comments

Comments in header files are meant for the users of classes and functions, while comments in implementation files are meant for those who maintain the classes.

Comments are often said to be either strategic or tactical. A strategic comment describes what a function or section of code is intended to do, and is placed before the code. A tactical comment describes what a single line of code is intended to do. Unfortunately, too many tactical comments can make code unreadable. For this reason, comments should be primarily strategic, unless trying to explain very complicated code (i.e. one should avoid stating in a comment what is clearly stated in code).

1. Give sensible names to types and variables rather than using obscure names that require additional comments to explain. The best code is self-documenting.
2. Use // for comments.  
   Rationale: A single standard provides consistency throughout the code.
3. Form complete sentences with proper capitalization and punctuation when writing documenting comments. Write other comments as complete sentences with proper capitalization and punctuation or as fragments beginning with lowercase letters and having no punctuation. Always use proper spelling and grammar.  
   Rationale: A single standard provides consistency throughout the code.
4. Delete code that is not used (commented out).  
   Rationale: No dead code is allowed.
5. Describe the externally visible behavior of the functions or classes being documented in header files.  
   Rationale: Exposing the internal workings of functions or classes to clients might enable those clients to create dependences on the internal representations.
6. Avoid stating in comments what is better stated in code (i.e. do not simply repeat what is in the code).  
   Rationale: While redundant comments are unnecessary, they also serve to increase the maintenance effort.  
   Example 1: The following example illustrates an unnecessary comment.  
   a = b + c; // Bad: add b to c and place the result in a.

Example 2: Avoid comments labeling obvious constructs as such. Anyone familiar with C++ will realize that this is a destructor.  
// Destructor (don’t do this)  
~MyClass();

1. Document every source file with an introductory comment that provides information on its author(s), its contents, and any program-required information (e.g. legal statements, copyright information, etc). Repeat this introductory comment in both header and implementation files if applicable.
2. Document assumptions (limitations) made by functions in the function’s preamble.  
   Rationale: Maintenance efforts become very difficult if the assumptions (limitations) upon which functions are built are unknown.
3. Place two spaces between the end of the code and the beginning of a comment for comments on the same line as code.  
   Rationale: Readability.  
   Example: return; // error already logged

## Declarations and Definitions

1. Do not use the same name for identifiers in an inner scope as an identifier in an outer scope, and therefore hiding that identifier.  
   Rationale: Hiding identifiers can be very confusing.
2. Make declarations at the smallest feasible scope.  
   Rationale: This guideline attempts to minimize the number of live variables that must be simultaneously considered. Furthermore, variable declarations should be postponed until enough information is available for full initialization (i.e. a variable should never be placed in a partly-initialized or initialized-but-not-valid state).
3. Make declarations at file scope static where possible.  
   Rationale: Minimize dependencies between translation units where possible. See AV Rule 137 in Appendix A of [1] for additional details.
4. Restrict identifiers to having either internal or external linkage within the same translation unit, not both.  
   Rationale: Avoid variable-name hiding, which can be confusing. See AV Rule 138 in Appendix A of [1] for further details.
5. Do not declare external objects in more than one file.  
   Rationale: Avoid inconsistent declarations. See AV Rule 139 in Appendix A of [1] for further details.  
   Note: This type of error will be caught by linkers, but typically later than is desired (i.e. the inconsistency could exist in a different group’s build). Normally this will mean declaring external objects in header files which will then be included in all other files that need to use those objects (including the files which define them).
6. Separate enumeration declarations and definitions.  
   Example:  
   enum Direction  
   {  
    Up,  
    Down  
   };  
   Direction direction;  
   Rationale: Readability. Naming the data type provides a mechanism to create other variables of the same type and allows type casting. See AV Rule 141 in Appendix A of [1] for more details.

## Initialization

1. Initialize all variables before use.  
   Rationale: Prevent the use of variables before they have been properly initialized. Exception: Exceptions are allowed where a name must be introduced before it can be initialized (e.g. value received via an input stream).
2. Do not introduce variables until they can be initialized with meaningful values.  
   Rationale: Prevent clients from accessing variables without meaningful values. See AV Rule 143 in Appendix A of [1] for examples.
3. Use braces to indicate and match the structure in the non-zero initialization of arrays and structures.  
   Rationale: Readability.  
   Example: int a[2][2] = { {0, 1}, {2, 3} };
4. In an enumerator list, do not use the ‘=’ construct to explicitly initialize members other than the first, unless explicitly initializing all items.  
   Rationale: Mixing the automatic and manual allocation of enumerator values is error-prone. Note that exceptions are allowed for clearly defined standard conventions. See AV Rule 145 in Appendix A in [1] for additional details.

## Types

1. Use enumeration types instead of integer types (and constants) to select from a limited series of choices.  
   Note: This rule is not intended to exclude character constants (e.g. ‘A’, ‘B’, ‘C’, etc.) from use as case labels.  
   Rationale: Enhances debugging, readability and maintenance.

## Constants

1. Do not use octal constants (other than zero).  
   Rationale: Any integer constant beginning with a zero (‘0’) is defined by the C++ standard to be an octal constant. Due to the confusion this causes, octal constants should be avoided.  
   Note: Hexadecimal numbers and zero (which is also an octal constant) are allowed.
2. Represent hexadecimal constants using all uppercase letters.
3. Use symbolic values rather than literal numeric values.  
   Rationale: Improved readability and maintenance.  
   Note: In many cases ‘0’ and ‘1’ are not magic numbers but are part of the fundamental logic of the code (e.g. ‘0’ often represents a null pointer). In such cases, ‘0’ and ‘1’ may be used.
4. Do not modify a string literal.  
   Rationale: The effect of attempting to modify a string literal is undefined  
   Note that strictly conforming compilers should catch violations, but many do not.

## Variables

1. Do not declare multiple variables on the same line. Rationale: Increases readability and prevents confusion.  
   Example:  
   int\* p, q; // Probably an error  
   int firstButtonOnTopOfLeftBox, i; // Bad: Easy to overlook i

## Unions and Bit Fields

1. Avoid using unions.  
   Rationale: Unions are not statically type-safe and are historically known to be a source of errors.  
   Note: In some cases, derived classes and virtual functions may be used as an alternative to unions.
2. Use only explicitly unsigned integral or enumeration types for bit-fields.  
   Rationale: Whether a plain (neither explicitly signed nor unsigned) char, short, int or long bit-field is signed or unsigned is implementation-defined. Thus, explicitly declaring a bit-filed unsigned prevents unexpected sign extension or overflow.
3. Do not use bit-fields to pack data into a word for the sole purpose of saving space.  
   Rationale: Bit-packing adds additional complexity to the source code. Moreover, bit packing may not save any space at all since the reduction in data size achieved through packing is often offset by the increase in the number of instructions required to pack/unpack the data.  
   Note: Bit-packing should be reserved for use in interfacing to hardware or conformance to communication protocols.  
   Warning: Certain aspects of bit-field manipulation are implementation-defined.

## Operators

1. Avoid side effect containing right hand operands with a && or || operator.  
   Rationale: Readability. The conditional evaluation of the right-hand side could be overlooked.
2. Parenthesize the operands of a logical && or || if the operands contain binary operators.  
   Rationale: Readability.
3. Do not overload ||, &&, and unary & operators.  
   Rationale: First, the behavior of the || and && operators depend on short-circuit evaluation of the operands. However, short-circuit evaluation is not possible for overloaded versions of the || and && operators. Hence, overloading these operators may produce unexpected results. Next, if the address of an object of incomplete class type is taken, but the complete form of the type declares operator&() as a member function, the resulting behavior is undefined.
4. Do not mix signed and unsigned values in arithmetic or comparison operations.  
   Rationale: Mixing signed and unsigned values is error prone as it subjects operations to numerous arithmetic conversion and integral promotion rules.
5. Avoid unsigned arithmetic.  
   Rationale: Over time, unsigned values will likely be mixed with signed values thus violating the previous guideline.
6. Do not apply the unary minus operator to an unsigned expression.
7. Do not use the sizeof operator on expressions that contain side effects.  
   Rationale: Clarity. The side effect will not be realized since sizeof only operates on the type of an expression: the expression itself will not be evaluated.
8. Prefer sizeof(varname) to sizeof(type).  
   Rationale: sizeof(varname) will update appropriately if someone changes the variable type either now or later.
9. Do not use the comma operator.  
   Rationale: Readability. The comma operator can be used to create confusing expressions. See AV Rule 168 in Appendix A of [1] for additional details.

## Pointers and References

1. Prefer references to pointers when possible.  
   Rationale: Pointers can be null, which demands additional consideration when dereferencing a pointer. Also the ‘.’ syntax usable with references is more concise than the ‘->’ syntax used with pointers.
2. Avoid pointers to pointers when possible.  
   Rationale: Pointers to pointers are a source of bugs and result in obscure code. Containers or some other form of abstraction should be used instead.
3. Avoid more than two levels of pointer indirection.  
   Rationale: Multiple levels of pointer indirections typically produce code that is difficult to read, understand and maintain.
4. If using C++11, use nullptr when comparing or assigning pointers. If not using C++11, use the NULL macro.
5. Use a typedef to simplify program syntax when declaring function pointers.  
   Rationale: Improved readability. Pointers to functions can significantly degrade program readability.
6. Avoid pointer arithmetic.  
   Rationale: The runtime computation of pointer values is error-prone (i.e. the computed value may reference unintended or invalid memory locations).  
   Exceptions: Objects such as containers, iterators, and allocators that manage pointer arithmetic through well-defined interfaces are acceptable. Pointer arithmetic is also allowed if deemed necessary to meet a performance goal.
7. Prefer smart pointers rather than handling raw pointers. Use smart pointers with semantics that correspond to how the pointer is used (e.g. unique\_ptr or scoped\_ptr when ownership of a pointer is limited to a single scope, shared\_ptr if multiple clients need a reference to a single object, etc.).

## Type Conversions

1. Avoid user-defined type conversions.  
   Rationale: User-defined conversion functions may be called implicitly in cases where the programmer may not expect them to be called. See AV Rule 177 in Appendix A of [1] for additional details.
2. Do not use implicit conversions that may result in a loss of information.  
   Rationale: The programmer may be unaware of the information loss. See AV Rule 180 in Appendix A of [1] for examples.
3. Do not type cast from any type to or from pointers.  
   Rationale: This type of casting can lead to undefined or implementation-defined behavior (e.g. certain aspects of memory alignments are implementation-defined). Furthermore, converting a pointer to an integral type can result in the loss of information if the pointer can represent values larger than the integral type to which it is converted.
4. Take every possible measure to avoid type casting.  
   Rationale: Errors caused by casts are among the most pernicious, particularly because they are so hard to recognize. Strict type checking is your friend – take full advantage of it.
5. Do not convert floating-point numbers to integers.  
   Rationale: Converting a floating-point number to an integer may result in an overflow or loss of precision.  
   Exceptions: Such conversions are permitted if necessitated by an algorithmic requirement or a hardware interface.
6. Use C++ style casts (const\_cast, reinterpret\_cast, dynamic\_cast, and static\_cast) instead of the traditional C-style casts.  
   Rationale: C-style casts are more dangerous than the C++ named conversion operators since the C-style casts are difficult to locate in large programs and the intent of the conversion is not explicit (i.e. (T) e could be a portable conversion between related types, a non-portable conversion between unrelated types, or a combination of conversions). See AV Rule 185 in Appendix A of [1] for additional details.

## Flow Control Structures

1. Remove all unreachable code.
2. Ensure all non-null statements potentially have a side effect.  
   Rationale: A non-null statement with no potential side effect typically indicates a programming error. See AV Rule 187 in Appendix A of [1] for additional information.
3. Do not use labels, except in switch statements.  
   Rationale: Labels are typically either used in switch statements or are as the targets for goto statements.
4. Do not use goto statements.  
   Rationale: Frequent use of the goto statement tends to lead to code that is both difficult to read and maintain.
5. Include a final else clause with all if, else if constructs or provide a comment indicating why a final else clause is not necessary.  
   Rationale: Provide a defensive strategy to ensure that all cases are handled by an else if series.  
   Note: This rule only applies when an if statement is followed by one or more else ifs.
6. Avoid fall through in a switch statement by terminating every non-empty case clause with a break statement. Consider moving the common code to a separate function and repeating the function call in each case clause.  
   Rationale: Eliminates potentially confusing behavior since execution will fall through to the code of the next case clause if a break statement does not terminate the previous case clause.  
   Exceptions: If a large proportion of the case clauses lead to the same code, omitting break statements is allowed but very clear comments indicating the fall through behavior are required.
7. Include a final default clause in all switch statements that do not intend to test for every enumeration value.  
   Rationale: Omitting the final default clause allows the compiler to provide a warning if all enumeration values are not tested in a switch statement. Moreover, the lack of a default clause indicates that a test for every case should be conducted. On the other hand, if all cases are not tested for, then a final default clause must be included to handle those untested cases.
8. Do not use a switch expression to represent a Boolean value.  
   Rationale: An if statement provides a more natural representation.
9. Include at least two cases and a potential default in every switch statement.  
   Rationale: An if statement provides a more natural representation.
10. Do not use floating-point variables as loop counters.  
    Rationale: Subjects the loop counter to rounding and truncation errors.
11. Avoid performing actions other than initializing the value of a single for loop parameter in the initialization expression in a for loop.  
    Note that the initialization expression may invoke an accessor that returns an initial element in a sequence:  
    for (Iter\_type p = c.begin() ; p != c.end() ; ++p)  
    {  
     …  
    }  
    Rationale: Readability.
12. Avoid performing actions other than changing a single loop parameter to the next value for the loop in the increment expression in a for loop.  
    Rationale: Readability.
13. Do not use null initialize or increment expressions in for loops; use a while loop instead.  
    Rationale: A while loop provides a more natural representation.
14. Do not modify numeric variables used within a for loop for iteration counting in the body of the loop.  
    Rationale: Readability and maintainability.

## Expressions

1. Do not test floating-point variables for exact equality or inequality.  
   Rationale: Since floating-point numbers are subject to rounding and truncation errors, exact equality may not be achieved, even when expected.
2. Do not allow evaluation of expressions to lead to overflow/underflow (unless required algorithmically and then should be heavily documented).  
   Rationale: Expressions leading to overflow/underflow typically indicate overflow error conditions.
3. Avoid placing several operations with side effects in a single expression.

Rationale: Readability. See AV Rule 204 in Appendix A of [1] for examples.

1. Ensure the value of an expression will remain the same under any order of evaluation that the standard permits.  
   Rationale: Except where noted, the order in which operators and subexpression are evaluated, as well as the order in which side effects take place, is unspecified. See AV Rule 204.1 in Appendix A of [1] for examples.

## Memory Allocation

1. Avoid un-encapsulated global data.  
   Rationale: Global data is dangerous since no access protection is provided with respect to the data.  
   Note: If multiple clients require access to a single resource, that resource should be wrapped in a class that manages access to that resource. For example, semantic controls that prohibit unrestricted access may be provided (e.g. singletons and input streams). See AV Rule 207 Appendix A of [1] for examples.

## Error Handling

1. Use exceptions rather than return values to communicate error conditions.  
   Rationale: Using exceptions for error handling makes code simpler, cleaner, and less likely to contain errors.

## Efficiency Considerations

1. Do not attempt to prematurely optimize code.  
   Rationale: Early focus on optimization can result in sacrificing the clarity and generality of modules that will not be the true bottlenecks in the final system.  
    *Premature optimization is the root of all evil –* Donald Knuth  
   Note: This rule does not preclude early consideration of fundamental algorithmic and data structure efficiencies.

## Miscellaneous

1. Prefer compile-time and link-time errors to run-time errors.  
   Rationale: Errors detected at compile/link time will not occur at run time.  
   Whenever possible, push the detection of an error back from run-time to link-time, and preferably compile-time.
2. Fully utilize compiler warning levels. Treat warnings as errors.  
   Rationale: Compilers can typically be configured to generate a useful set of warning messages that point out potential problems. Information gleaned from these messages could be used to resolve certain errors before they occur at runtime.

# References

1. Joint Strike Fighter Air Vehicle C++ Coding Standards. Doc. No. 2RDU00001 Rev C. http://www.stroustrup.com/JSF-AV-rules.pdf
2. Google C++ Style Guide. https://google-styleguide.googlecode.com/svn/trunk/cppguide.html