C++ Coding Standards

Revised: November 14, 2006

1. Purpose

The purpose of this document is to provide consistent coding guidelines for computer programs coded in the C++ programming language for the OpenEaagles project.

2. Document Organization

This document is organized into rules, reasons, and recommendations. Rules are things that the programmer must do, and will be checked for at peer reviews. Reasons explain why it should be done that way. Recommendations are suggestions, and not taking the suggestion will not be considered an error at a peer review.

3. Source Code Organization

Rule: Use the standard file extensions listed in the table below.

Extension	Description
.c	C Source files
.cpp	C++ Source files
.h	C/C++ Header files

Reason: Consistency and clarity.

Rule: When a single class is in a file, name the file according to the class being defined.
Reason: This should make the program more understandable and confirmable.

Rule: When a single class is in a file, filename capitalization should match classname capitalization..
Reason: This should make the program more understandable and confirmable. In Microsoft Windows, the file names “System.h” and “system.h” are seen as the same file,

so file names cannot be unique only by character case. While MS Windows is case insensitive in its file management system, it does display the file names with the proper character case.

Rule: Do not include tabs in source files; use spaces instead.
Reason: Tab settings vary with operating systems and editors, which hinders open system development and readability.

Rule: Do not use deprecated functions (e.g., strcpy(), strcat(), strstream, std::copy(), the old C I/O library, etc.).
Reason: Conforms to the current C++ language standards.

Rule: Use relative, not absolute, paths to refer to header files in "include" statements.
Reason: Using absolute paths makes portability and maintenance more difficult.

Rule: Do not use "pragma" (e.g., turning off warnings) or "using namespace" statements in header files.
Reason: These will affect the user's code and can produce undesired side effects.

Rule: Guarding preprocessor directive code must be added to every header file.
Example:

```
#ifndef Sample_header
#define Sample_header
…rest of header file…
#endif //Sample_header
```

Reason: This avoids multiple inclusions of header files.

Rule: Keep the coding and documentation styles consistent within a file.
Reason: Consistency aids in readability. Different programmers have different preferences on this issue, but when modifying a file, stay consistent with the form that is already there.

Rule: Header files should contain the documentation for it's class(es). This should include the class name, description and purpose, events and slots handled, and a description of all public member functions.
Reason: Ensures that at a minimum, well-documented header files (and code) are provided to the users. All other required documentation (e.g., user guides, interface descriptions, etc.) are in addition to the documented header files.

Rule: Document the public interface to the class at the top of the header file (*.h). Document private/protected variables in the header file and document private/protected functions in the source file (*.cpp).
Reason: This rule will make the program more readable.

Rule: When overriding a base class' virtual member functions, declare these at the end of each section (public, protected, private) with the proceeding comment "XXXX Class Interface", where XXX is the base class that first declared the virtual functions.
Reason: Code readability. This identifies that we're overriding a base class' interface.

Recommendation: Use C++ style comments, "// comment", in C++ code. Use C style comments, "/* comment */", in C code and for embedded comments.
Example:

```
// C++ style for comment blocks
int n = 5; // and for in-line notes at the end of a statement
for ( int i = 0; i < n; i++ /* C style for embedded comment */ ) { }
```

Reason: Many C++ projects will include a mixture of C++ and C files. Using C++ and C style comments helps to identify the language within the file. However, C++ comments do not allow for embedded comments; they continue to the end of the line. Also, C code is restricted by the language to only C style comments.

4. Variables

Rule: Start all variable names and function names with a lower case letter, and start each additional word in the name with a capital letter.
Examples: startWithLowerCase, otherWordsGetUpperCase.
Reason: This will aid in program readability.

Recommendation: Variable names should be descriptive.
Reason: This should make the program more understandable and confirmable.

Rule: Declare variables using the smallest possible scope.
Reason: Improves readability and partitioning of code.

Rule: Do not use global variables or constants.
Reason: This improves reliability. There is no control over who can modify global variables.

Rule: Initialize all variables to an initial value before use.
Reason: Default initial values vary with compilers.

Rule: Declare only one variable per statement.
Reason: This improves readability and reliability by making it easier to add qualifiers, initialize, and comment the variable.

```
const int iconst = 15;   // Example single line declaration
```

Rule: Initialize pointers to a valid address or to zero when initially created.
Reason: This will make the program more reliable.

Rule: Declare variables when they are first assigned a value
Example:

```
A* p = new A();
int k = i + j + 5;
```

Reason: Satisfies the rules on initialization, limiting scope, and one variable per statement.

Rule: In ”for” loops, limit the scope of the indices to the loop.
Example 1:

```
for (int i = 0; i < n; i++) {
   /* scope of ‘i’ is limited to the loop */
}
```

Example 2:

```
int i;
for (i = 0; i < n; i++) {
   /* scope of ‘i’ is NOT limited to the loop */
}
```

Reason: This limits the indices to the smallest possible scope. Note that some compilers require setting an option switch to enable “In For Loop Scope”.

Rule: If a pointer is passed into a function, that function must verify it is not zero.
Reason: This will make the program more reliable.

Rule: Do not use ”void” pointers in a function prototype.
Reason: This increases reliability; void pointers can not be type checked.

Rule: Use ”true” and ”false” with ”bool” type variables for logical operations instead of “TRUE” and “FALSE” with ”int” type variables.
Reason: Type checking

Rule: Set unused or invalid pointers to zero instead of “NULL”.
Reason: “NULL” is a holdover from C, and although it is usually set to zero anyway, this may not always be the case. You will produce undesired results when you pass a “NULL” pointer, which has not been defined as zero, to a function that is expecting either a “zero” pointer.

Rule: When testing non-boolean (bool) types with a conditional statement, do an explicit comparison with zero instead of relying on the compiler’s implicit test with zero.
Examples:

```
if (p != 0)  // Correct
if (p)       // Incorrect
if (p == 0)  // Correct
if (!p)      // Incorrect
```

Reason: The result of a conditional operation should be determined by an explicit logical expression or a boolean variable.

Recommendation: When using the “new” operator to allocate dynamic memory, always check to see whether “new” returns the zero pointer.
Reason: It is possible that the system won’t have enough memory to satisfy the request. In this case, “new” returns the value zero.

Recommendation: Use “L” for the suffix instead of “l” when creating a long.
Reason: The small “l” looks like a one, and using the capital “L” makes the code easier to read.

Rule: When declaring a pointer or reference, attached the “*” or “&” to the type.

Example:

```
int*  ptr = new int;  // memory allocated on free store
```

Reason: Pointer and reference qualifiers change the variable's type, not its name. In addition, this is required to apply "const" to a pointer being passed in a function call. Example: func(const int* const p)

Rule: When allocating memory for a pointer, use the keyword "new", and not the old "malloc()".
Reason: This is C++

Rule: After calling "delete" or "unref()" on a pointer, set it to zero.
Reason: This prevents dangling pointers.

Rule: When passing values or objects into functions, use a "const" reference both for values that won't change ("in") and for pointers for objects that will be returned ("in-out").
Reason: When using pointers to pass values by reference, the repeated need to dereference the pointers within the function is a reminder that an external value is being changed.

Rule: Name constants using all capitalized characters and, if needed, underscores. For example: MAX_ARRAY_LENGTH.
Reason: This increases readability.

Rule: Variable names based on acronyms should not contain all capitalized characters.
Reason: Does not clearly standout as a variable name and could be confused with a constant name (see above). For example, the variable for Time-Of-Flight should be "tof" and not "TOF".

Rule: Declare constants as "const" variables or as enumerated type, and not with preprocessor "#define" directive.
Reason: The preprocessor's #define directive does a global substitution, which can produce unwanted side effects, and it does not provide for the kind of type checking that is provided by the compiler for variables and enumerated types.

Rule: When throwing an exception, throw a pointer to the exception.
Example:

```
throw new ExpWeHaveAProblem();
```

Reason: An exception can be copied several times before it is caught. Copying just the pointer is more efficient.

Rule: Use "dynamic_cast" to cast pointers and test the resulting pointer for zero.
Reason: Using "dynamic_cast" checks the type info of the object being cast and returns zero if incorrect. Note that some compilers require that you set an option for run-time type checking.

Rule: Don't use platform specific type definitions. For instance, "long long".
Reason: This hinders portability.

Rule: Do not cast-away the ”const” qualifier
Reason: This removes the compiler’s ability to insure that a constant variable or object is not changed.

Rule: Do not create a new object in an argument in a function call
Example: func(new Abc())
Reason: This will create a memory leak. It’s ambiguous who is responsible for unreferencing or deleting the object.

5. Classes

Rule: Start all class and type names with an upper case letter, and start each additional word in the name with a capital letter.
Example: ClassNamesStartUpperCase
Reason: This will aid in program readability.

Rule: Class names based on acronyms should not contain all capitalized characters.
Reason: Does not clearly standout as a class name and could be confused with other naming conventions. For example, a Surface-to-Air-Missile class should be named ”Sam” and not ”SAM”, which could be confused with a constant.

Rule: When creating classes, arrange the contents so that the public part will come first, then the protected, and then the private.
Reason: This makes the code more understandable and uniform.

Rule: Do not use public member variables.
Reason: This rule supports the engineering principles of information hiding and modularity.

Recommendation: Do not use protected member variables.
Reason: This conforms with the engineering principles of information hiding and modularity.

Rule: Do not use multiple inheritance.
Reason: Use of multiple inheritance increases maintenance and program complexity.

Rule: Add the “const” qualifier to all pointers or references returned by member functions that point to internal, private data.
Example: const float* getDataArray() const;
Reason: This will restrict the ability to modify private data, making the program more reliable.

Rule: If there is a requirement to return a pointer without the “const” qualifier in a member function, then provide both a const and non-const version of the function.
Example:

```
float* getDataArray();
```

```
const float* getDataArray() const;
```
Reason: When a “const” qualifier is added to an instance of the object the second function is used by the compiler.

Rule: All non-static member variables must be initialized by the constructor or a function called by the constructor, and all static member variables must be initialized in the *.cpp file.
Reason: This rule will make the program more reliable.

Rule: Do not initialize a static member variable from another class’ static member variable.
Reason: We can not insure that the static member variables are initialized in the correct order.

Rule: When using a pointer to a class, don’t include the class’ header file. Instead declare it with an incomplete type declaration.
Example:
```
#include “A.h”; // Not needed (This is what not to do)
Class A;        // Incomplete type declaration (Do this)
Class B;
{
   <some code>
   <some more code>
private
   A* pa;       // a pointer to class A.
}
```
Reason: This aids in the principle of information hiding.