Observations Report

Drew Cabales

Challenge: Technical Review

National Weather Service/OHD
Science Infusion and Software Engineering Process Group (SISEPG) – C++ Coding Standards and Guidelines Peer Review Checklist

C++ Coding Standards and Guidelines Peer Review Checklist

Last Updated: 25 April 2016

Reviewer's Name:		**Peer Review Date:**	
Project Name:		**Project ID:** Enter if applicable	
Developer's Name:		**Project Lead:**	
Review Files & Source code			
Code Approved			

The following check list is to be used in the assessment of C++ source code during a peer review. Items which represent the code being reviewed should be checked.

1. General Programming Standards and Guidelines

Refer to the OHD General Programming Standards and Guidelines Peer Review Checklist to assess the adherence to the OHD General Programming Standards and Guidelines.

2. C++ Programming Standards

2.1 Readability and Maintainability

__✓__ Consistent indentation (3 or 4 spaces)

__✓__ Consistent use of braces

__✗__ No tabs used

2.2 File Names

__✓__ Header files and namespace files use suffixes: `.h`, `.H`, `.hh`, `.hpp`, or `.hxx`

__✓__ Source files use suffixes: `.C`, `.cc`, `.cpp`, or `.cxx`

__✓__ `UpperMixedCase` is used for class or namespace file names

__✗__ `lowerMixedCase` is used for function file names

2.3 File Organization

✓ Each file contains only one class declaration or definition except functors and static classes

✓ File includes a brief description of the file after the *documentation block*

✗ The content of the file is in the following order:

1. The preprocessor directives to prevent multiple inclusions in header files.
2. The *Documentation block* described in the *"OHD General Software Development Standards and Guidelines"*
3. A brief description of the file
4. Include files
5. `#defines` and Macros
6. The '`use`' directives in the source files but not in header files
7. Class or function declaration or definition

2.4 Include Files

✓ C++ standard library headers that have no extension are used

✗ New prefix `c` is used instead of the old extension `.h` for C standard header files

✓ The < > pair for library and system headers is used

✗ The " " pair for non-system (user defined) headers is used

✓ No absolute or relative paths to point to the header files are used

✓ The system header files first in alphabetical order followed by the non system include files (including COTS includes) also in alphabetical order

2.5 Comments

✓ The JavaDoc convention format is used for the documentation comment

✗ The C++ comment "//" style or the C style (/* ... */) is used for inline comments

2.6 Naming Schemes

✓ `namespace`, `class`, `struct`, template argument, and parameter names use uppercase letters as word separators with the first character capitalized

✓ Macro and `#defined` constant, `enum`, `union`, class static data member, and global variable names are all capitalized with underscore as separators

✗ Class methods and variable names use uppercase letters as word separators with the first character is not capitalized

✓ Private class data member names are prepended with the underscore, the rest is

the same as method names

___X___ `static const` data members are all uppercase

___√___ `typedef` names reflect the style appropriate to the underlying type

___√___ Class, struct, variable, and method names that differ by case only are not used

___√___ C function names follow the *OHD C Programming Standards and Guidelines*

2.7 Class Design

___√___ Class members are declared in this order: public members, protected members, private members

___X___ Data members are properly protected (declared as private or protected)

___√___ Classes (except functors and static classes) implement a default constructor, a virtual destructor, a copy constructor, and an overloaded assignment operator

___√___ Static classes declare a private default constructor to prevent instantiation

2.8 Safety and Performance

___√___ Type conversions have been done explicitly. The C++ set of casting operators `static_cast`, `reinterpret_cast`, `const_cast` and `dynamic_cast` have been used instead of C-style casting

___√___ Global variables are not used except in rare cases and when used include an inline comment describing the reason for use.

___√___ Dynamically allocated memory is deallocated when no longer needed

___√___ There is no dangling pointers. Pointers are always tested for `NULL` values before trying to dereference them

___√___ There is no hardcoded numerical values, const or enum type values are used instead

___√___ Large objects are created on the heap

___√___ The arguments specified in a function prototype are associated with variable names

3. C++ Programming Guidelines

3.1 Readability and Maintainability

___√___ A space is put between the parenthesis and the keywords or the function names

___√___ A space is put between variables, keywords and operators

___√___ Pointers are named in some fashion that distinguishes them from other "ordinary" variables

___√___ Parentheses are used in macros to ensure correct evaluation of the macro

___✓___ The `goto` statement is used very sparingly

3.2 User Defined Types

___✓___ `static const` members are used instead of `#defined` constants

___✓___ Proper `typedefs` are used instead of using templates directly

___✓___ `enum` is used to define a collection of integral constants

3.3 Variables

___✓___ `const` correctness has been practiced

___✗___ All variables are correctly initialized

___✓___ Local variables are declared near their first use.

___✓___ The copy constructor is used to construct an object instead of the assignment operator (=)

3.4 Performance

___✓___ `inline` functions are used instead of Macros

___✓___ The prefix form (`++i` or `--i`) is used instead of postfix form (`i++` or `i--`)

___✓___ Pointer arithmetic has been avoided

___✓___ Repetitive computations are reduced by only doing them once and saving the result in a temporary variable for future access

3.5 Class Design

___✓___ Parts-of relation inheritance has been avoided

Questions:

1. In the Code File of the challenge, there was a potential for a major memory leak in the “UberNode”. What was it?
   UberNode’s deconstructor isn’t called when it’s deleted

2. Static Testing encapsulates a couple software testing methods that include Static Analysis and Code Reviews. Static testing is automated testing that checks for code validation without executing the code. Static Testing also covers Code Reviews, which is done by software developers and engineers to find what errors are occurring, and how often. They look into potential mitigations for their vulnerabilities and record what errors are common within their codebase.

3. We can write tests that can conclude there’s a chance that a specific section of code has errors based on common design patterns and implementations of classes and libraries, but to prove that it’s correct or not is too complex for an automated system to detect.

I learned about the core differences between static and dynamic testing, and how both are important to produce clean, efficient code that will hopefully include minimal bugs. Since there’s a long list of well known exploits that are common amongst many companies' code in the past and present, these static analysis tools are important in recognizing flaws that software engineers might not have knowledge of. There’s a limit to automated testing, however it proves to be a good defense measure after human analysis and the code compiler.