PRQA Programming Research

WHITE PAPER

HICPP, JSF++ and MISRA C++: a study of rule overlaps and effective compliance

By Wojciech Basalaj, Senior Technical Consultant

November 2011

Any organization wishing to adopt best practices for its C++ development needs to consider the coding rules to prescribe in its Coding Standard. There are many publicly available guidelines to use as a starting point, with the 3 most comprehensive being High Integrity C++ (HICPP), Joint Strike Fighter Air Vehicle C++ (JSF++) and MISRA C++. While there appear to be preferences in certain industries for following one of these guidelines over all the others, the rationale is often only an emotive one.
This article aims to shed some light on overlaps between HICPP, JSF++ and MISRA C++, with the intention of helping in the aforementioned coding standard formulation process, as well as understanding the effort of complying with more than one off-the-shelf coding standard, for example in cases where code is reused and different coding standards are mandated contractually.

Overlaps between C++ Coding Standards
A comparison of HICPP, JSF++ and MISRA C++

By Wojciech Basalaj, Senior Technical Consultant

Introduction

The principle of Coding Standards has long been an emotive one among software developers, with attitudes ranging from "why do we need such restrictions?" to "how can we possibly operate without such controls?"

Their primary objective is to prevent unwanted behavior or misbehavior of software. Software languages generally contain features that, in their entirety, are rich beyond the needs of most software practitioners. What this means is that most ordinary developers are not expected to be expert in the full language feature-set, and coding rules help to protect them against areas of language danger or misuse.

While the already well-publicized and documented undefined behaviors of these languages are central to any language protection, there are many other types of vulnerability that require deep understanding of language syntax and semantics. Close analysis of the transformation from raw source code to object and executable image is just as important in achieving a high quality and robust code base.

It is clear that the eradication of undefined behavior problems from a code base during the development cycle using a comprehensive set of rules (a coding standard) will have a positive impact on the quality and cost of the software developed. Prohibiting the use of dangerous parts of the language and dangerous practices, via an automatic enforcement solution (static analysis tool) is the most effective way to achieve this on current code and future development practices.

In general terms, a coding standard's effectiveness can be measured by the degree of automatic enforceability. Manual code inspection has been proven to be ineffectual, costly, time consuming and prone to human error. A key criteria of a coding standard and any process certification that decrees the use of one, is that the coding standard should be enforced by an automated tool.

HICPP Background

The High Integrity C++ (HICPP) standard [9] was first introduced in 2003 by Programming Research and can be freely obtained from **www.codingstandard.com**. It is based on topics from C++ literature [1] [2] [3] [4] [5] [6] [7] alongside best practices in C++ development. The standard defines a set of rules for the production of high quality C++ code. An explanation is provided for each rule. Each rule shall be enforced unless a formal deviation is recorded for which it outlines a deviation process. The guiding principles of the standard are maintenance, portability, readability and safety and its aim is to arrive at high quality source code by ascertaining that the code complies with these principles. The standard adopts the view that restrictions should be placed on the ISO C++ language [2] in order to limit the flexibility it allows. This approach has the effect of minimizing problems created either by compiler diversity, different programming styles, or dangerous/confusing aspects of the language. Different compilers may implement only a subset of the ISO C++ standard or interpret its meaning in a subtly different way that can lead to porting and semantic errors. Without applying good standards, programmers may write code that is prone to bugs and/or difficult for someone else to pick up and maintain.

JSF AV++ Background

The JSF AV++ coding standard [10] produced by Lockheed Martin™ is intended to help programmers develop code that conforms to safety critical software principles, i.e. code that does not contain defects that could lead to catastrophic failures resulting in significant harm to individuals and/or equipment.

Overall, the philosophy embodied by the rule set is essentially an extension of C++'s philosophy with respect to C constructs. That is, by providing "safer" alternatives to "unsafe" facilities, known problems with low-level features are avoided. In essence, programs are written in a "safer" subset of a superset. In general, the code produced should exhibit the following important qualities:

- have a consistent style,
- be portable to other architectures,
- be free of common types of errors, and be understandable, and hence maintainable, by different programmers.

The purpose is to define a C++ rule set to produce code that is more correct, reliable, and maintainable.

Rules are required for Air Vehicle C++ development and recommended for non-Air Vehicle C++ development and were designed to specifically address unpredictable behavior:

- Restrict programmers to a better specified, more analyzable, and easier to read (and write) subset of C++
- Eliminate large groups of problems by attacking their root causes (e.g. passing arrays between functions as pointers)
- Ban features with behaviors that are not 100% predictable (from a performance perspective)

The AV Coding Standard, although originally intended for aviation/aerospace, is now more widely adopted by a number of industries and promotes, monitors and controls:

- Reliability: Executable code should consistently fulfill all requirements in a predictable manner.
- Portability: Source code should be portable (i.e. not compiler or linker dependent).
- Maintainability: Source code should be written in a manner that is consistent, readable, simple in design, and easy to debug.
- 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: code size, complexity and static path count (number of paths through a piece of code)
- Reusability: The design of reusable components is encouraged. Component reuse can eliminate redundant development and test activities (i.e. reduce costs).
- Extensibility: Requirements are expected to evolve over the life of a product. (i.e. perturbations in requirements may be managed through local extensions rather than wholesale modifications).
- Readability: Source code should be written in a manner that is easy to read, understand and comprehend.

MISRA C++ Background

The automotive industry has been using C for a number of years but the requirements in infotainment have necessitated the use of C++ to gain greater flexibility and usability. MIRA decided that this would need to be investigated with a mission “to provide assistance to the automotive industry in the creation and application of safe and reliable software in vehicle systems”

During discussion on the viability of creating a safety critical standard for C++ it was felt that this committee should link with the thoughts of the Automotive industry requirement as some of the people on the MISRA C Committee were also involved in the proposed committee all be it of a different skill set (C++).

The MISRA C++™ guidelines [11] form a set of restrictions and a safe subset of the C++ language suitable for the development of safety critical systems and other embedded applications. The standard draws from established coding standards such as MISRA C[8], Lockheed Martin’s Joint Strike Fighter Air Vehicle C++ coding standard (JSF++) and PRQA’s High-Integrity C++ coding standard (HICPP), and is intended to contain a set of guidelines and best practices understandable to a wide audience. MISRA with a prime requirement for safety critical use being predictability, aims to

- Promote best practice in automotive safety-related systems engineering
- Develop guidance in specific technical areas such as the C++ language, software readiness for production and safety analysis
- Eliminate or mitigate unpredictability
  - unspecified behavior – it is simply not known what the program will do
  - implementation dependent – different behavior on different platforms
  - unknown execution time
  - unknown resource requirements
- Improve clarity for review and maintenance.
- Provide a consistent style across a program or set of programs
- Avoid common programmer errors
- Incorporate good practice, particularly with regard to ‘future proofing’.

Public input was sought and received as a means of public review of the suggested guidelines.

Coding Standards Overlap

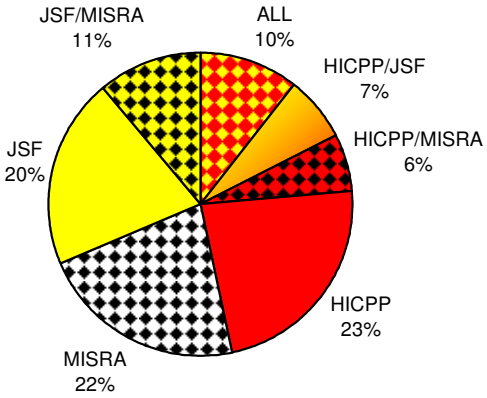

Figure 1

The complete rule set spanning the HICPP, JSF++ and MISRA C++ coding standards contains 431* unique rules, which is considerably less than the sum total of the rules standing at 663, due to significant overlaps. As can be seen from Figure 1, each coding standard contains nearly a half of that unique rule set*, with approximately equal split between rules unique to each coding standard, and rules shared with at least one other standard. Just 45 rules (10%) appear in all coding standards; further 106 rules appear in 2 coding standards simultaneously.

The 45 common rules can be traced to shared references:
ISO/IEC 14882:2003, Programming Languages - C++ [2]
MISRA C:2004 [8]
Industrial Strength C++ [6]
Scott Meyers' Effective C++ [3]
Bjarne Stroustrup's The C++ Programming Language [1]
and some rules that first appear in HICPP [9]

These probably constitute the absolute minimum that any tailored C++ coding standard should contain. An example of such a common rule is Item 21 of Effective C++ which advocates using const whenever possible. Interestingly, all but one of these common rules are amenable to static analysis, and they are available in QA·C++ [12].

* Because granularity of rules varies among coding standards, e.g. with 1 rule corresponding to a few in another standard, this number is open to some interpretation. * HICPP slices are denoted with red hue, JSF++ with yellow, and MISRA C++ with chequered pattern

On the other end of the scale, the rules that appear in only a single coding standard suffer from a few drawbacks:

- these significantly outnumber the common rules, which makes their selection and potential enforcement time consuming
- a high proportion of the rules - 52% - cannot be enforced with static analysis, and thus need to be enforced through other means, e.g. manual code review.

For example, a concept that is unique to MISRA C++ is that of underlying types, which it inherits from MISRA C:2004. HICPP is unique with its guidance for the use of STL in Chapter 17. A rule that is unique to JSF is Rule 103 which prescribes to apply constraint checks to template arguments.

The rules that are shared between two coding standards are not that numerous and are more susceptible to static analysis – only 24% have to be manually enforced. An example of a rule common to JSF and MISRA C++ only is that arrays should not be passed as parameters, to avoid the "array decay" problem. HICPP and MISRA C++ have in common that polymorphic member functions have to be declared virtual explicitly in every derived class. HICPP and JSF share that they allow public derivation only.

Rule Enforcement

Coding standard rules may be impossible to enforce statically for a number of reasons. Some rules are documentation or process based rather than dependent on source code, and as such cannot be automated; good examples are MISRA C++ Rule 1-0-2 'Multiple compilers shall only be used if they have a common, defined interface' and JSF++ Rule 218 'Compiler warning levels will be set in compliance with project policies'. Moreover, coding standards sometimes contain rules that are too vague or high level to enforce automatically, e.g. MISRA C++ Rule 15-0-1 'Exceptions should only be used for error handling' and JSF++ Rule 216 'Programmers should not attempt to prematurely optimize code'. It may be possible to enforce certain aspects of such rules once the underlying intent is clarified.

Further differences in enforceability typically occur on the basis of a rule being considered mandatory or advisory in a coding standard. The rationale of classifying the rules as such is that non compliance to a mandatory rule is typically considered to carry more risk in terms of program correctness, as opposed to breaking an advisory rule. HICPP

partitions its rule set into 'Required' and 'Advisory' rules; JSF++ contains 'Shall' and 'Will' rules which are mandatory and 'Should' rules are advisory; finally MISRA C++ takes a slightly different approach with 'Required' and 'Document' rules which are mandatory and with the remaining rules being labeled as 'Advisory'. Subsequently, we will refer to the rules from either of these coding standards as mandatory or advisory.

Table 1: Manual Enforcement

	mandatory	advisory
HICPP	23%	53%
JSF++	17%	47%
MISRA C++	23%	6%
MISRA C++ B	20%	33%

As can be seen from Table 1 a higher proportion of advisory rules can only be enforced manually than is the case for mandatory rules. The third row of the table suggests that MISRA C++ may have the opposite trend; however, on closer inspection the discrepancy is caused by classifying 'Document' rules as mandatory. The great majority of these rules are process based and can only be enforced manually. The final row of the table details enforcement figures when 'Document' rules are considered together with 'Advisory' rules, and it follows the same trend as for HICPP and JSF++.

Rule Selection

Some organizations address the cost/benefit considerations of adding extra rules to their coding standard by segregating rules into severity levels. Legacy or non-production code may only need to adhere to the first tranche of rules. The next levels are applicable only to new code or legacy code once an 'amnesty' period is over. Each off-the-shelf coding standard already supports this by categorising rules into mandatory and advisory. This can be refined further, using subjective judgment, or better with objective techniques, like the classifications based on rule overlaps and automated enforcement.

We would recommend the classification based on rule overlap between off-the-shelf coding standards to be given the highest importance when considering rules for inclusion in a tailored coding standard, on the basis that following widely agreed best practice is desirable. Thereafter, we see the mandatory attribute and automatic enforcement as equally important; hence mandatory, statically enforceable rules are distinctly favored over advisory and manually enforced rules, with the other two combinations taking a middle ground.

Legacy Code Management

Key aspect in the adoption of a coding standard is that it is introduced in an incremental fashion so as not to adversely disrupt existing code and existing practices, and to ensure the successful buy in of the revised development environment and quality processes. MISRA C++ ([11] Section 4.3.4) itself recognizes and recommends that the requirements of the standard are introduced in a progressive manner and may take in the region of 1-2 years to implement all aspects of the document

It is important to identify and differentiate between different stages of code that will be subjected to a coding standard. Applying a subset of the guidelines to "legacy code" will ensure that progressive improvement of existing code does not adversely affect the quality and behavior, whilst applying a more comprehensive set of rules to "new" code will ensure the best conformance level. To achieve effective compliance this process should be supported by a static analysis tool directly, as illustrated in Figure 2. Alternatively, organizations with large bodies of existing code, can utilize a "baselining" (legacy code management) technique to ensure that all new code or changes to legacy code conform to a full set of guidelines, whereas the legacy or untouched code remains unaffected. Over time as legacy code is maintained and reused, an increasing proportion of code lines will be touched, and hence made to comply with the full coding standard. However, the key benefit is that this will be done in a piecemeal and non-intrusive fashion.

Figure 3 shows in more detail how a baseline is generated for the first version of the project source and used for subsequent versions of the project. It also shows a necessary management mode which a quality supervisor can use to control the suppressions that are allowed. They can choose to remove suppressions for violations that are perceived dangerous or add suppressions corresponding to formal deviations to coding standard rules. In this way, the system provides an implementation for a formal deviation process which is an important aspect of coding standard enforcement.

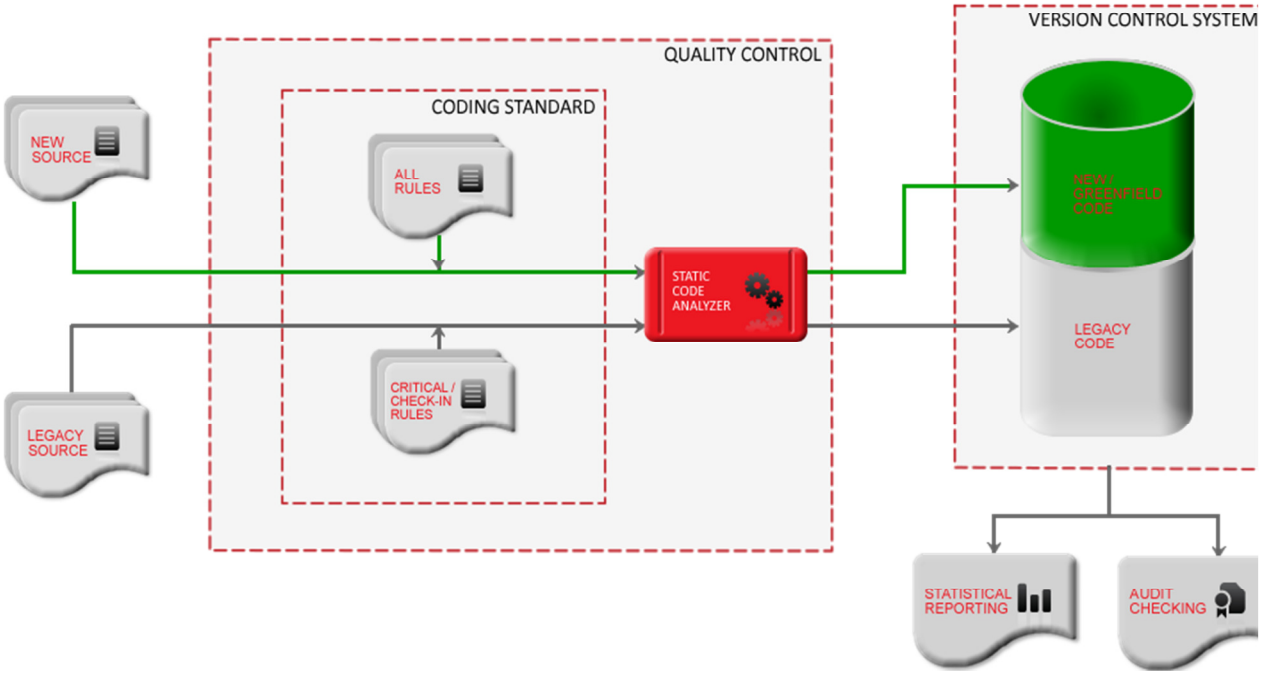

Figure 2: Legacy Code versus New Code Rule Application

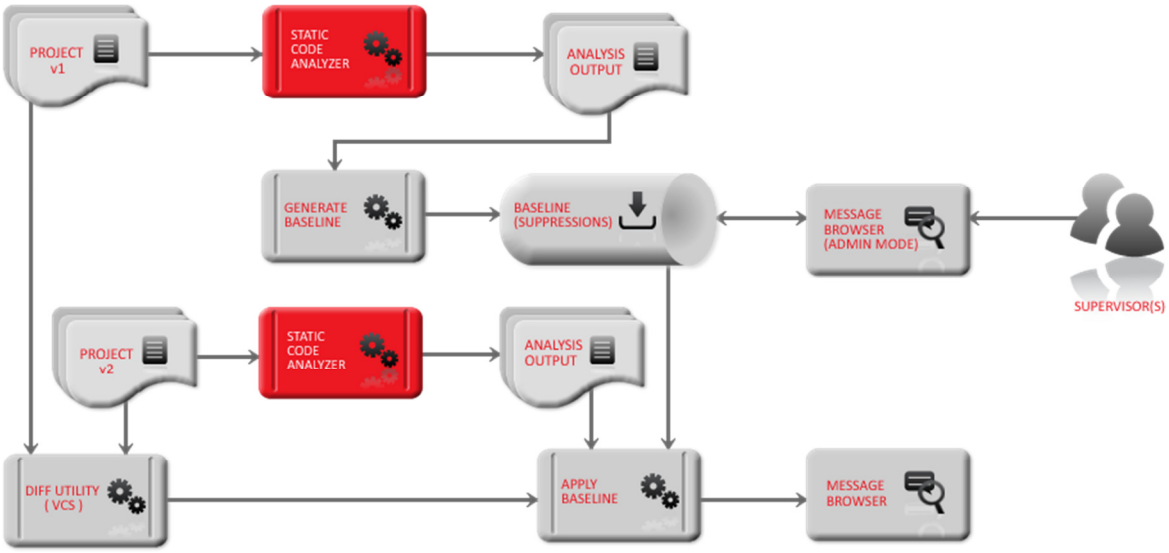

Figure 3: Baselining System Architecture

Summary

We provided an overview of three popular C++ coding standards originating from different industries and investigated their overlap. When tailoring a coding standard for a given organization we recommend to at least include all the common rules, and also to favor mandatory and statically enforceable rules over advisory or manually enforced rules. Finally we suggested a "baselining" approach to gradually introduce coding standard enforcement into a development process, accommodating legacy or 3rd party code.

References

[1] Bjarne Stroustrup, "The C++ Programming Language". Addison-Wesley. 2000

[2] International Standard ISO/IEC 14882:2003 "Programming Language C++"

[3] Scott Meyers, "Effective C++", Addison-Wesley. 1996

[4] Scott Meyers, "More Effective C++", Addison-Wesley. 1996

[5] Scott Meyers, "Effective STL", Addison-Wesley. 2001

[6] Mats Henricson, Erik Nyquist, Ellemtel Utvecklings AB, "Industrial Strength C++", Prentice Hall. 1997

[7] Herb Sutter, "Exceptional C++", Addison-Wesley. 2000

[8] Motor Industry Research Association, "MISRA-C:2004 - Guidelines for the use of the C language in critical systems", ISBN 0 9524156 2 3, , October 2004

[9] Programming Research, "High Integrity C++ Coding Standard Manual - Version 3.2", http://www.codingstandard.com, 2008

[10] Lockheed Martin, "Joint Strike Fighter Air Vehicle C++ Coding Standards For The System Development And Demonstration Program", Document Number 2RDU00001 Rev C, December 2005

[11] Motor Industry Research Association, "MISRA-C++:2008 - Guidelines for the use of the C++ language in critical systems, 2008

[12] Programming Research, "QA·C++ Static Source Code Analyzer", http://www.programmingresearch.com, 2009

Contact Us

Programming Research Ltd.
Mark House
9/11 Queens Road
Hersham
Surrey
KT12 5LU
United Kingdom

Tel: +44 1932 888 080
Fax: +44 1932 888 081
info@ProgrammingResearch.com
www.programmingresearch.com