PRQA Programming Research

WHITEPAPER

High Integrity C++

www.codingstandard.com

High Integrity C++ Coding Standard V4.0 - an overview

By Wojciech Basalaj, Richard Corden

November 2013

On 3rd October 2013, High Integrity C++ (HIC++), one of the most respected, longest established and widely adopted C++ coding standards, marked its 10th anniversary with the publication of a major new update – Version 4.0. Over the past decade more than 24,000 copies of this coding standard have been downloaded.

In this whitepaper the co-authors of HIC++ provide an overview of HIC++ V4.0, specifically:

- *the rationale behind the V4.0 release, including C++11 semantics and associated good practices*
- *the fit with other key C++ coding standards, such as MISRA C++, JSF AV C++ and CERT C++*
- *the key updates compared to the prior versions of HIC++*
- *the practical implications for development teams transitioning to V4.0*

1 Introduction

The original High Integrity C++ coding standard, published on 3rd October 2003, pooled best practice advice available at the time [Stroustrup][Effective C++][More Effective C++][Effective STL][Industrial Strength C++][Exceptional C++], as well as internal know-how at PRQA. The result was a set of 202 semantic and syntactic (and specifically excluding stylistic) rules and guidelines, setting up a safe subset of the C++ language [ISO C++ 2003].

In the intervening decade other publicly available language subsets emerged [JSF AV C++][MISRA C++], which specifically focus on use of C++ for safety critical applications, and both of which reused many of the original HIC++ rules. All these coding standards share an approach of defending against dangerous, confusing or non-portable language constructs, thus can be termed language subsets.

The recently ratified C++ language updates [ISO C++ 2011] invalidate some of the existing advice, and expose gaps relating to the use of new constructs such as lambdas, rvalue references and new facilities of the standard library. In order to prevent HIC++ from losing its relevance, this major revision addresses these updates.

2 Rule Alterations

The relationship between versions 3.3 and 4.0 of HIC++ is summarized in Figure 1, and further explained in subsequent sections.

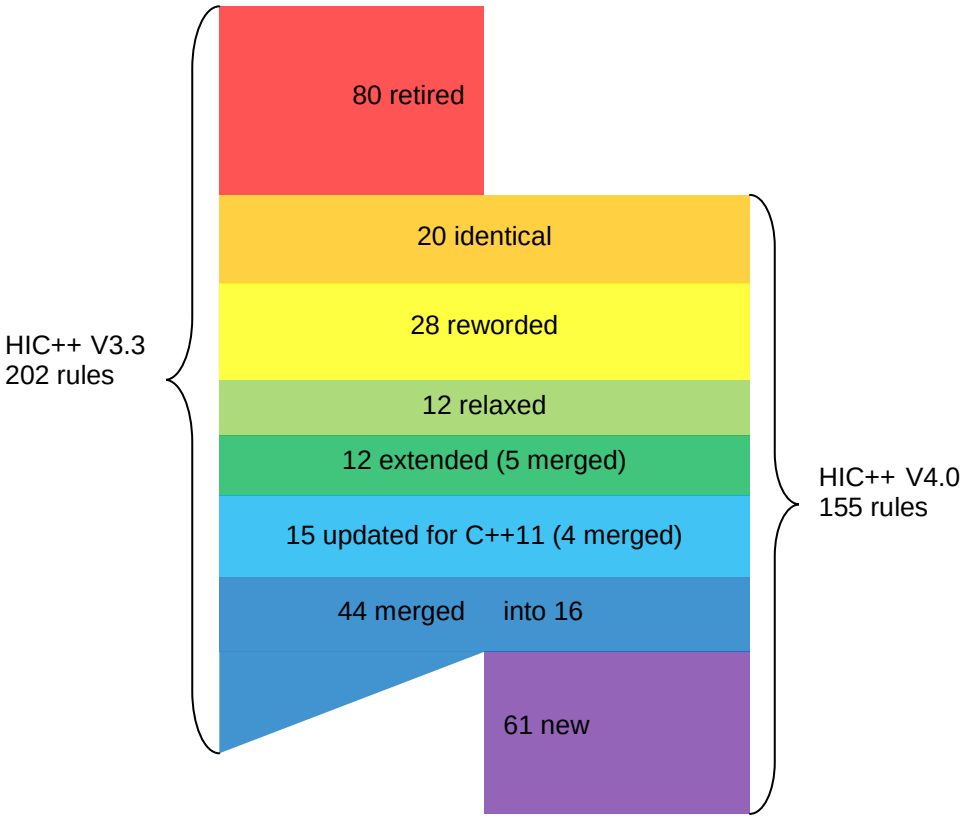

Figure 1: Composition of HIC++ V3.3 and V4.0.

2.1 Retired Rules

When a coding standard contains too many rules, there can be rule overlaps and other enforcement problems. HIC++ V4.0 retired 80 rules for various specific reasons with the overall objective of making the rule-set more enforceable and manageable.

Some rules were not consistent with the guiding principles of HIC++: code maintenance, portability, readability and robustness. A good example is the retired guideline 3.1.7, which is simply too subjective and without a proven case for or against following it:

retired	Do not use the 'inline' keyword for member functions, inline functions by defining them in the class body.

Other retired rules were not amenable to automated enforcement, thus limiting their practical value, for example rule 3.3.7:

retired	Only define virtual functions in a base class if the behavior will always be valid default behavior for derived classes.

Finally, a small number of rules have become obsolete due to improvements in language specification in the new C++ standard [ISO C++ 2011], as is now the case for guideline 10.18:

retired	Guard the modulus operation to ensure that both arguments are non-negative.

2.2 General & Specific Rule Overlap

A common idiom in a coding standard is to have a general avoidance rule, e.g. 'minimize the use of casts', and another constraining its use, e.g. 'avoid using pointer or reference casts'. This introduces confusion and redundancy. A single rule is preferable - typically the more restrictive one - with a rule deviation mechanism for localized non-compliance.
Rules with similar rationale but pertaining to different features, represent another aspect of redundancy in HIC++ V3.3. A good example is the following pair of rules:

Rule 8.4.2 (V3.3)	Declare each variable on a separate line in a separate declaration statement...
Rule 8.4.7 (V3.3)	Declare one type name only in each typedef declaration.

The key consideration is their common rationale, and we decided to merge several such cases into a single rule.
In some cases, rules dealing with a problematic C++ feature were able to be retired in favor of a modern coding idiom. For example, the following new rule consolidates advice on pointers and resource handles, addressing exception safety, and preventing memory leaks and dangling pointers:

3.4.3 (V4.0)	Use RAII for resources

Overall, by following these principles we managed to replace 44 overlapping rules with 16 more generic ones, as detailed in an Appendix of the HIC++ manual [HIC++4].

2.3 C++11 Updates

In the latest version of the C++ standard [ISO C++ 2011], new syntax has been added to simplify the definition of special member functions, namely `=delete` and `=default`, rendering obsolete rules relating to the declaration of copy constructors, copy assignment operators and destructors:

Rule 3.1.3 (V3.3)	Declare or define a copy constructor, a copy assignment operator and a destructor for classes which manage resources.
Guideline 3.1.13 (V3.3)	Verify that all classes provide a minimal standard interface against a checklist comprising: a default constructor; a copy constructor; a copy assignment operator and a destructor.

These rules have been replaced in HIC++ V4.0 with one with clearer text and much simplified enforcement:

12.5.1	Define explicitly =default or =delete implicit special member functions of concrete classes

Similarly, special identifiers final and override allow simplification of rules relating to overridden virtual member functions. Additionally, the new standard library header <cstdint> defines* size specific type aliases, e.g. int32_t, which are preferable to similar type definitions in user code, as advocated by previous best practice†. In all, 15‡ rules have been updated based on introduction of better alternatives in C++11.

2.4 Other Significant Modifications

In a few cases, namely 12 rules, we felt that extending the rules, by making them in effect more restrictive, would benefit their justification and enforceability. As an example:

Rule 9.2 (V3.3)	Only throw objects of class type.
15.1.1 (V4.0)	Only use instances of std::exception for exceptions

Conversely, 12§ other rules were relaxed, as we believed them to be overly restrictive, without a good case matching the guiding principles of HIC++, as detailed in Section 2.1. As an example:

Rule 11.3 (V3.3)	Specify the name of each function parameter in both the function declaration and the function definition. Use the same names in the function declaration and definition.
8.2.1 (V4.0)	Make parameter names absent or identical in all declarations

3 New Rules

In total, 61 new rules have been added, covering topics detailed below. This brings the total rule count in HIC++ V4.0 to 155, down from 202 in V3.3. Each rule now has a comprehensive explanation with examples of compliance and non-compliance, and is categorized using its relevant clause and sub-clause from the text of the language standard [ISO C++ 2011]. The new categorization ensures that related rules are grouped, and allows for easy navigation within the document, as well as cross referencing with the language standard.

3.1 Prior Language Subsets

In a separate whitepaper [PRQA Overlaps], overlaps between publicly available language subsets [HIC++3][JSF AV C++][MISRA C++] were examined. In particular, it was found that several rules are common to JSF AV C++ and MISRA C++ but absent from HIC++ V3.3. Clearly, these represented potential gaps in the language subset. After review, 15 rules have been incorporated into the new version of HIC++, which are heavily influenced by their JSF AV C++ and MISRA C++ counterparts. One of the rules has also been updated to fit the new language specification, as elaborated in Section 2.3. In addition, 6 rules unique to MISRA C++ have also been incorporated, with one updated to match C++11.

Notably, we have borrowed from JSF AV C++ the notion of an *interface class*, and defined it in HIC++ as follows:

- all public functions are pure virtual functions or getters, and
- there are no public or protected data members, and
- it contains at most one private data member of integral or enumerated type

Addition of this concept simplifies formulation of a rule limiting use of multiple inheritance:

10.3.1 (V4.0)	Ensure that a derived class has at most one base class which is not an interface class

* if available in a particular compiler implementation
† HIC++ V3.3 Rule 8.4.6, JSF AV C++ Rule 209, MISRA C++ Rule 3-9-2
‡ also 4 have been merged as detailed in Section 2.2
§ also 5 have been merged as detailed in Section 2.2

An analysis of rule overlap between HIC++ and MISRA C++/JSF++ standards reveals an increase in rules common or unique to HIC++ (55% from 46% previously), and a decrease in exclusively MISRA C++/JSF++ rules. Figure 2 presents a side-by-side comparison of this rule overlap. In order to perform an accurate comparison, the effects of rule merges and deletions are ignored.

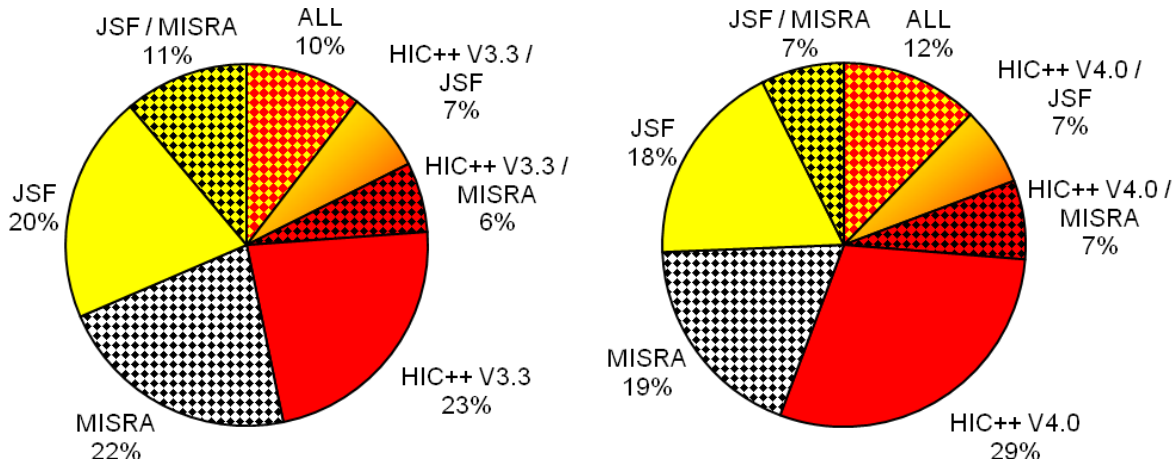

Figure 2: Overlaps between HIC++, JSF AV C++ and MISRA C++. HIC++ slices are denoted with red hue, JSF++ with yellow, and MISRA C++ with checkered pattern.

CERT C++ Secure Coding Standard [CERT C++] is another commonly used coding standard. As this is the most recent entrant to the C++ coding standards arena, it shares many rules with the prior coding standards. A considerable proportion of the remaining rules are not concerned with the use of C++ language features, e.g. rules constraining use of the POSIX API or focused on code behavior. For this reason it is not strictly speaking a subset of the C++ language. Only two CERT C++ rules are quoted in HIC++ V4.0, with the remaining rules being redundant (as discussed in Section 2.2), previously covered elsewhere, or not directly related to the C++ language.

PRQA is actively involved in various C++ forums [WG21][ACCU], where on occasion C++ vulnerabilities are discussed. Through such unpublished sources, we have identified 4 additional rules not specific to C++11 for inclusion in the new version of HIC++. These rules can be identified in the HIC++ Manual [HIC++4] by the absence of any external references.

3.2 C++11

The new version of C++ standard [ISO C++ 2011] has revised the language definition substantially, and in particular has added several new features, e.g. lambdas and rvalue references. Mainstream C++ compilers have already implemented these features, so it is important to address dangerous aspects of their use. C++ experts have started disseminating guidance on the use of these features [GOTW] [Effective C++ '11], and we have based 7 new rules on this material. In addition, we have added 17 rules originating from unpublished sources, as described in Section 3.1, and our independent analysis of the text of the language standard [ISO C++ 2011].

3.3 Concurrency

Previously, ISO C++ lacked any support for multithreading and synchronization. This has been rectified in the latest version of the standard [ISO C++ 2011], by explicitly providing data race guarantees, and extending the standard library with the provision of the Atomic Operations and Thread Support libraries. Based on published expert sources [Sutter Hardware][Williams Concurrency] we have formulated 12 rules. Notably, we have provided an additional class high_integrity::thread, which wraps and enforces correct usage of std::thread, and provided its definition in the following rule:

18.2.1 (V4.0)	Use high integrity::thread in place of std::thread

4 Enforcement in Practice

So far we have provided justification for all the rule changes and additions in HIC++V4.0. However, an open question remains as to how these modifications affect the enforceability of HIC++, one of the most popular language subsets for C++ [VDC]

In order to examine overall compliance levels, we analyzed several open source projects initially with QA·C++ (PRQA's Static Analysis tool) and the HIC++ Compliance Module V3.3 and then, after suppressing all non-compliances, re-analyzed using QA·C++ and HIC++ V4.0 Compliance Module to present the unique V4.0 non-compliances.

Table 1 lists all these open source projects, and column 3 displays their corresponding deterioration in compliance. On aggregate, for projects that already comply with HIC++ V3.3, the additional burden in fully[**] migrating to the latest version is modest, at 15% of the original effort of compliance.

		Ratio of violation differences for HIC++ versions (%)	
project	**KLOC**	**Additional in V4.0**	**Retired from V3.3**
cgicc-3.2.9	11	23%	9%
commoncpp2-1.8.1	45.5	28%	24%
cvc3-2.4.1	5.5	33%	22%
doxygen-1.8.1.2	51.2	11%	17%
libsigc++-2.2.10	24.6	38%	8%
pstoedit-3.60	40.4	16%	10%
QScintilla-gpl-2.6.2	106	11%	16%
re2c-0.13.5	12	12%	21%
ucommon-5.2.2	68.3	21%	25%
znotes-0.4.5	12.3	14%	15%
total	376.8		
Average Ratio		**15%**	**17%**

Table 1: Comparison of violations unique to HIC++ V4.0 (3rd column) and V3.3 (4th column). The averages were calculated by treating the whole collection of projects as a single codebase.

To complete this study we inverted the HIC++ versions and reanalyzed the codebases, as detailed above. This has allowed us to see if the adjustment for retired rules (as described in Section 2.1), overlapping rules (as per Section 2.2), and relaxed rules (as per Section 2.4) improves the ability to achieve overall compliance with V4.0. Column 4 demonstrates that HIC++ V4.0 avoids 17% of unnecessary violations on average. Therefore, on balance the new version of HIC++ requires less effort to enforce, while constituting a more current and up-to-date language subset, as detailed in Sections 2 and 3.

5 Comparison with the MISRA Guidelines

In March 2013, the latest version of the MISRA C guidelines was published [MISRA C:2012]. This is the third edition of the guidelines which cover the use of C in critical systems. Questions may arise as to the difference in safety approach of HIC++ V4.0 and MISRA, any pure differences in rule content, as well as the absence of the MISRA C:2012 meta concepts of rules versus directives, "decidability" and analysis scope.

[**] Full compliance is a challenging target at best, and a more realistitic approach is to gradually achieve compliance using agreed deviation from selected rules.

5.1 Application Domain Differences

Many of the rules selected for HIC++ share roots with rules in MISRA, and a common approach to reliability and safety can be observed. Where HIC++ differs slightly from MISRA is due to target domain considerations. MISRA C:2012 targets software for use in critical embedded systems. In such environments the cost of failure, no matter how rare, is significant, and equally the operating domain is highly constrained. Rules need to handle a greater range of failure cases.

For example, MISRA C:2012 Dir 4.12 disallows the use of dynamic memory. This is appropriate within a critical embedded environment where an attempt to acquire more memory than is available would result in the failure of the system. Similarly, as per MISRA C:2012 Rule 21.6, the use of I/O in critical applications is disallowed because its behavior can be unspecified or undefined. However, I/O is a requirement for many applications in non-critical domains and so such a rule cannot be generally applied.

HIC++ is designed principally to operate in a less constrained system software environment, where system resources can reasonably expect to be acquired, external access including I/O is a given, and exception handling techniques can accommodate a wider array of failure modes.

5.2 Rules vs Directives

MISRA C:2012 explicitly distinguishes between issues that can be detected through analysis of source code alone (“Rules”), and items that must be verified from an external source (“Directives”) and thus documented outside of the source code domain.

The rule set of HIC++ has been selected and written in such a way that enforcement should be possible through source code analysis. For example, the HIC++ rule relating to assembler usage is worded such that any use of the 'asm' declaration is a violation of the rule:

7.5.1	Do not use the asm declaration

5.3 Decidable vs Demonstrable

The MISRA C:2012 guidelines include the concept of a rule being ‘decidable’, when it is theoretically possible for a static analysis tool to determine if code complies to the rule in every case.

HIC++ adopts a slightly different approach, introducing the concept of demonstrability, requiring that the issue is not possible in the source code:

4.2.2	Ensure that data loss does not demonstrably occur in an integral expression
5.2.1	Ensure that pointer or array access is demonstrably within bounds of a valid object
5.5.1	Ensure that the right hand operand of the division or remainder operators is demonstrably non-zero

6 Conclusions

This whitepaper presents genealogy of the latest version of HIC++ as reviewed HIC++ V3.3, new rules and rule revisions specific to C++11, additional rules common to JSF++ and MISRA C++ not already in HIC++, and also concurrency rules. As a key consideration, rules are formulated to avoid conflicts or overlap, in order to improve enforceability. Rule presentation has been improved, with a new categorization based on the relevant clause and sub-clause of the C++ language standard, and a comprehensive justification with examples. The principles of the latest HIC++ and MISRA C guidelines have been discussed and the differences noted, the foremost of which is the applicability of HIC++ to any development domain. Finally, it has been shown that the rule changes have beneficial effect on enforceability, with improved signal-to-noise ratio, when compared to the previous version of HIC++.

References

[ACCU]	http://www.accu.org/
[CERT C++]	CERT C++ Secure Coding Standard, https://www.securecoding.cert.org
[Effective C++]	Scott Meyers: Effective C++, 1996, Addison-Wesley
[More Effective C++]	Scott Meyers: More Effective C++, 1996, Addison-Wesley
[Effective C++ '11]	Scott Meyers: Draft TOC for Effective C++11 Concurrency Chapter, http://scottmeyers.blogspot.hu/2013/04/draft-toc-for-ec11-concurrencychapter.html
[Effective STL]	Scott Meyers: Effective STL, 2001, Addison-Wesley
[Exceptional C++]	Herb Sutter: Exceptional C++, 2000, Addison-Wesley
[GOTW]	Herb Sutter: Guru of the Week, http://herbsutter.com/gotw/
[HIC++3]	High Integrity C++ Coding Standard Manual - Version 3.2, http://www.codingstandard.com, October 2008, Programming Research
[HIC++4]	High Integrity C++ Coding Standard Manual - Version 4.0, http://www.codingstandard.com, October 2013, Programming Research
[Industrial Strength C++]	Mats Henricson, Erik Nyquist, Ellemtel Utvecklings AB: Industrial Strength C++, 1997, Prentice Hall
[ISO C++ 2003]	International Standard ISO/IEC 14882:2003(E) Programming languages – C++.
[ISO C++ 2011]	International Standard ISO/IEC 14882:2011(E) Information technology – Programming languages – C++.
[JSF AV C++]	Joint Strike Fighter Air Vehicle C++ Coding Standards Rev C, December 2005, Lockheed Martin Corporation
[MISRA C++]	MISRA C++:2008 Guidelines for the use of the C++ language in critical systems, June 2008, MIRA Limited
[MISRA C:2012]	MISRA C:2012 Guidelines for the use of the C language in critical systems, March 2013, MIRA Limited
[PRQA Overlaps]	HICPP, JSF++ and MISRA C++: a study of rule overlaps and effective compliance, November 2011, Programming Research, http://www.programmingresearch.com/resources/white-papers/
[Stroustrup]	Bjarne Stroustrup: The C++ Programming Language. Addison-Wesley. 2000
[Sutter Hardware]	Herb Sutter: The C++ Memory Model and Modern Hardware, http://herbsutter.com/2013/02/11/atomic-weapons-the-c-memory-modeland-modern-hardware
[VDC]	The Increasing Value and Complexity of Software Call for the Reevaluation of Development and Testing Practices, April 2011, VDC, http://www.programmingresearch.com/resources/white-papers/
[WG21]	ISO/IEC JTC1/SC22/WG21 - The C++ Standards Committee, http://www.open-std.org/jtc1/sc22/wg21/
[Williams Concurrency]	Anthony Williams: C++ Concurrency in Action - Practical Multithreading, 2012, Manning Publications Co.

About the Authors

Wojciech Basalaj

Wojciech is a senior developer at PRQA, primarily responsible for the Dataflow analysis component of QA·C and QA·C++. Previously, he looked after pre- and post-sales support of PRQA's products. Through hands on and customer facing experience he amassed knowledge of C++ coding standards, and is a co-author of High Integrity C++.

Richard Corden

Richard is a lead developer at PRQA, primarily responsible for the enhancement of QA·C++, a static analysis tool for the C++ language. He has extensive experience in the development and implementation of C++ coding standards, and is a co-author of MISRA C++ and High Integrity C++. He represents PRQA at the ISO C++ Committee [WG21] meetings.

About PRQA/Programming Research

Established in 1985, PRQA, ISO 9001 and TickIT certified, is recognized throughout the industry as a pioneer in static analysis, championing automated coding standard inspection and defect detection, delivering its expertise through industry-leading software inspection and standards enforcement technology used by over 3,000 companies globally.

PRQA's industry-leading tools, QA·C, QA·C++ and QA·Verify, offer the closest possible examination of C and C++ code. All contain powerful, proprietary parsing engines combined with deep accurate dataflow which deliver high fidelity language analysis and comprehension. They identify problems caused by language usage that is dangerous, overly complex, non-portable or difficult to maintain. Plus, they provide a mechanism for coding standard enforcement.

PRQA has corporate offices in UK, USA, India, Ireland and Netherlands, complemented by a worldwide distribution network. Find out more at **www.programmingresearch.com**

Contact Us

Email: info@programmingresearch.com
Web: www.programmingresearch.com