PRQA
Programming Research

High Integrity C++

Coding Standard
Version 4.0

www.codingstandard.com

3 October 2013

Programming Research Ltd
Mark House - 9/11 Queens Road - Hersham - Surrey KT12 5LU - United Kingdom
Tel: +44 (0) 1932 888 080 - Fax: +44 (0) 1932 888 081
info@programmingresearch.com - www.programmingresearch.com
Registered in England No. 2844401

Contents

0 Introduction **7**
0.1 Typographical Conventions . . . 7
0.2 Escalation policy . . . 7
0.3 Base Standard and Policy . . . 8
0.3.1 ISO Standard C++ . . . 8
0.3.2 Statically detectable restrictions . . . 8
0.3.3 Examples . . . 8
0.4 Basis of requirements . . . 8
0.5 Rule Enforcement . . . 8
0.6 Deviations . . . 9
0.7 Glossary . . . 9

1 General **10**
1.1 Implementation compliance . . . 10
1.1.1 Ensure that code complies with the 2011 ISO C++ Language Standard . . . 10
1.2 Redundancy . . . 10
1.2.1 Ensure that all statements are reachable . . . 10
1.2.2 Ensure that no expression or sub-expression is redundant . . . 11
1.3 Deprecated features . . . 12
1.3.1 Do not use the increment operator (++) on a variable of type `bool` . . . 12
1.3.2 Do not use the `register` keyword . . . 12
1.3.3 Do not use the C Standard Library .h headers . . . 13
1.3.4 Do not use deprecated STL library features . . . 13
1.3.5 Do not use `throw` exception specifications . . . 14

2 Lexical conventions **15**
2.1 Character sets . . . 15
2.1.1 Do not use tab characters in source files . . . 15
2.2 Trigraph sequences . . . 15
2.2.1 Do not use digraphs or trigraphs . . . 15
2.3 Comments . . . 16
2.3.1 Do not use the C comment delimiters `/* ... */` . . . 16
2.3.2 Do not comment out code . . . 17
2.4 Identifiers . . . 18
2.4.1 Ensure that each identifier is distinct from any other visible identifier . . . 18
2.5 Literals . . . 18
2.5.1 Do not concatenate strings with different encoding prefixes . . . 18
2.5.2 Do not use octal constants (other than zero) . . . 19
2.5.3 Use `nullptr` for the null pointer constant . . . 19

3 Basic concepts **21**
3.1 Scope . . . 21
3.1.1 Do not hide declarations . . . 21
3.2 Program and linkage . . . 21
3.2.1 Do not declare functions at block scope . . . 21
3.3 Storage duration . . . 22
3.3.1 Do not use variables with static storage duration . . . 22
3.4 Object lifetime . . . 23
3.4.1 Do not return a reference or a pointer to an automatic variable defined within the function . . . 23
3.4.2 Do not assign the address of a variable to a pointer with a greater lifetime . . . 24

3.4.3 Use RAII for resources . . . 25
3.5 Types . . . 27
3.5.1 Do not make any assumptions about the internal representation of a value or object . . . 27

4 Standard conversions **29**
4.1 Array-to-pointer conversion . . . 29
4.1.1 Ensure that a function argument does not undergo an array-to-pointer conversion . . . 29
4.2 Integral conversions . . . 29
4.2.1 Ensure that the `U` suffix is applied to a literal used in a context requiring an unsigned integral expression . . . 29
4.2.2 Ensure that data loss does not demonstrably occur in an integral expression . . . 30
4.3 Floating point conversions . . . 32
4.3.1 Do not convert an expression of wider floating point type to a narrower floating point type . . . 32
4.4 Floating-integral conversions . . . 32
4.4.1 Do not convert floating values to integral types except through use of standard library functions 33

5 Expressions **34**
5.1 Primary expressions . . . 34
5.1.1 Use symbolic names instead of literal values in code . . . 34
5.1.2 Do not rely on the sequence of evaluation within an expression . . . 34
5.1.3 Use parentheses in expressions to specify the intent of the expression . . . 35
5.1.4 Do not capture variables implicitly in a lambda . . . 36
5.1.5 Include a (possibly empty) parameter list in every lambda expression . . . 37
5.1.6 Do not code side effects into the right-hand operands of: `&&`, `||`, `sizeof`, `typeid` or a function passed to `condition_variable::wait` . . . 37
5.2 Postfix expressions . . . 39
5.2.1 Ensure that pointer or array access is demonstrably within bounds of a valid object . . . 39
5.2.2 Ensure that functions do not call themselves, either directly or indirectly . . . 40
5.3 Unary expressions . . . 41
5.3.1 Do not apply unary minus to operands of unsigned type . . . 41
5.3.2 Allocate memory using `new` and release it using `delete` . . . 41
5.3.3 Ensure that the form of `delete` matches the form of `new` used to allocate the memory . . . 42
5.4 Explicit type conversion . . . 43
5.4.1 Only use casting forms: `static_cast` (excl. `void*`), `dynamic_cast` or explicit constructor call 43
5.4.2 Do not cast an expression to an enumeration type . . . 44
5.4.3 Do not convert from a base class to a derived class . . . 45
5.5 Multiplicative operators . . . 46
5.5.1 Ensure that the right hand operand of the division or remainder operators is demonstrably non-zero . . . 46
5.6 Shift operators . . . 46
5.6.1 Do not use bitwise operators with signed operands . . . 46
5.7 Equality operators . . . 47
5.7.1 Do not write code that expects floating point calculations to yield exact results . . . 47
5.7.2 Ensure that a pointer to member that is a virtual function is only compared (`==`) with `nullptr` 47
5.8 Conditional operator . . . 48
5.8.1 Do not use the conditional operator (`?:`) as a sub-expression . . . 48

6 Statements **49**
6.1 Selection statements . . . 49
6.1.1 Enclose the body of a selection or an iteration statement in a compound statement . . . 49
6.1.2 Explicitly cover all paths through multi-way selection statements . . . 49
6.1.3 Ensure that a non-empty case statement block does not fall through to the next label . . . 50
6.1.4 Ensure that a switch statement has at least two case labels, distinct from the default label . . . 50

6.2 Iteration statements . . . 52
6.2.1 Implement a loop that only uses element values as a range-based loop . . . 52
6.2.2 Ensure that a loop has a single loop counter, an optional control variable, and is not degenerate 53
6.2.3 Do not alter a control or counter variable more than once in a loop . . . 54
6.2.4 Only modify a for loop counter in the for expression . . . 54
6.3 Jump statements . . . 55
6.3.1 Ensure that the label(s) for a jump statement or a switch condition appear later, in the same or an enclosing block . . . 55
6.3.2 Ensure that execution of a function with a non-void return type ends in a return statement with a value . . . 56
6.4 Declaration statement . . . 57
6.4.1 Postpone variable definitions as long as possible . . . 57

7 Declarations **59**
7.1 Specifiers . . . 59
7.1.1 Declare each identifier on a separate line in a separate declaration . . . 59
7.1.2 Use `const` whenever possible . . . 59
7.1.3 Do not place type specifiers before non-type specifiers in a declaration . . . 60
7.1.4 Place CV-qualifiers on the right hand side of the type they apply to . . . 61
7.1.5 Do not inline large functions . . . 61
7.1.6 Use class types or typedefs to abstract scalar quantities and standard integer types . . . 62
7.1.7 Use a trailing return type in preference to type disambiguation using `typename` . . . 63
7.1.8 Use `auto id = expr` when declaring a variable to have the same type as its initializer function call . . . 64
7.1.9 Do not explicitly specify the return type of a lambda . . . 65
7.1.10 Use `static_assert` for assertions involving compile time constants . . . 65
7.2 Enumeration declarations . . . 66
7.2.1 Use an explicit enumeration base and ensure that it is large enough to store all enumerators 66
7.2.2 Initialize none, the first only or all enumerators in an enumeration . . . 67
7.3 Namespaces . . . 68
7.3.1 Do not use *using directives* . . . 68
7.4 Linkage specifications . . . 68
7.4.1 Ensure that any objects, functions or types to be used from a single translation unit are defined in an unnamed namespace in the main source file . . . 68
7.4.2 Ensure that an inline function, a function template, or a type used from multiple translation units is defined in a single header file . . . 69
7.4.3 Ensure that an object or a function used from multiple translation units is declared in a single header file . . . 69
7.5 The asm declaration . . . 70
7.5.1 Do not use the `asm` declaration . . . 70

8 Definitions **71**
8.1 Type names . . . 71
8.1.1 Do not use multiple levels of pointer indirection . . . 71
8.2 Meaning of declarators . . . 71
8.2.1 Make parameter names absent or identical in all declarations . . . 71
8.2.2 Do not declare functions with an excessive number of parameters . . . 72
8.2.3 Pass small objects with a trivial copy constructor by value . . . 73
8.2.4 Do not pass `std::unique_ptr` by const reference . . . 73
8.3 Function definitions . . . 74
8.3.1 Do not write functions with an excessive McCabe Cyclomatic Complexity . . . 74
8.3.2 Do not write functions with a high static program path count . . . 75
8.3.3 Do not use default arguments . . . 76

8.3.4 Define =delete functions with parameters of type *rvalue reference* to const . . . 76
8.4 Initializers . . . 76
8.4.1 Do not access an invalid object or an object with indeterminate value . . . 76
8.4.2 Ensure that a braced aggregate initializer matches the layout of the aggregate object . . . 78

9 Classes **79**
9.1 Member functions . . . 79
9.1.1 Declare static any member function that does not require this. Alternatively, declare const any member function that does not modify the externally visible state of the object . . . 79
9.1.2 Make default arguments the same or absent when overriding a virtual function . . . 79
9.1.3 Do not return non-const handles to class data from const member functions . . . 80
9.1.4 Do not write member functions which return non-const handles to data less accessible than the member function . . . 81
9.1.5 Do not introduce virtual functions in a final class . . . 82
9.2 Bit-fields . . . 83
9.2.1 Declare bit-fields with an explicitly unsigned integral or enumeration type . . . 83

10 Derived classes **84**
10.1 Multiple base classes . . . 84
10.1.1 Ensure that access to base class subobjects does not require explicit disambiguation . . . 84
10.2 Virtual functions . . . 85
10.2.1 Use the override special identifier when overriding a virtual function . . . 85
10.3 Abstract classes . . . 86
10.3.1 Ensure that a derived class has at most one base class which is not an interface class . . . 86

11 Member access control **88**
11.1 Access specifiers . . . 88
11.1.1 Declare all data members private . . . 88
11.2 Friends . . . 89
11.2.1 Do not use friend declarations . . . 89

12 Special member functions **91**
12.1 Conversions . . . 91
12.1.1 Do not declare implicit user defined conversions . . . 91
12.2 Destructors . . . 91
12.2.1 Declare virtual, private or protected the destructor of a type used as a base class . . . 91
12.3 Free store . . . 92
12.3.1 Correctly declare overloads for operator new and delete . . . 92
12.4 Initializing bases and members . . . 93
12.4.1 Do not use the dynamic type of an object unless the object is fully constructed . . . 93
12.4.2 Ensure that a constructor initializes explicitly all base classes and non-static data members . . . 94
12.4.3 Do not specify both an NSDMI and a member initializer in a constructor for the same non static member . . . 95
12.4.4 Write members in an initialization list in the order in which they are declared . . . 96
12.4.5 Use delegating constructors to reduce code duplication . . . 96
12.5 Copying and moving class objects . . . 98
12.5.1 Define explicitly =default or =delete implicit special member functions of concrete classes . . . 98
12.5.2 Define special members =default if the behavior is equivalent . . . 99
12.5.3 Ensure that a user defined move/copy constructor only moves/copies base and member objects . . . 100
12.5.4 Declare noexcept the move constructor and move assignment operator . . . 101
12.5.5 Correctly reset moved-from handles to resources in the move constructor . . . 102

12.5.6 Use an atomic, non-throwing swap operation to implement the copy and move assignment operators . . . 103
12.5.7 Declare assignment operators with the ref-qualifier & . . . 105
12.5.8 Make the copy assignment operator of an abstract class protected or define it =delete . . . 106

13 Overloading **108**
13.1 Overload resolution . . . 108
13.1.1 Ensure that all overloads of a function are visible from where it is called . . . 108
13.1.2 If a member of a set of callable functions includes a universal reference parameter, ensure that one appears in the same position for all other members . . . 110
13.2 Overloaded operators . . . 111
13.2.1 Do not overload operators with special semantics . . . 111
13.2.2 Ensure that the return type of an overloaded binary operator matches the built-in counterparts 112
13.2.3 Declare binary arithmetic and bitwise operators as non-members . . . 112
13.2.4 When overloading the subscript operator (operator[]) implement both const and non-const versions . . . 113
13.2.5 Implement a minimal set of operators and use them to implement all other related operators . 114

14 Templates **116**
14.1 Template declarations . . . 116
14.1.1 Use variadic templates rather than an ellipsis . . . 116
14.2 Template instantiation and specialization . . . 116
14.2.1 Declare template specializations in the same file as the primary template they specialize. . . . 116
14.2.2 Do not explicitly specialize a function template that is overloaded with other templates . . . 117
14.2.3 Declare extern an explicitly instantiated template . . . 118

15 Exception handling **119**
15.1 Throwing an exception . . . 119
15.1.1 Only use instances of std::exception for exceptions . . . 119
15.2 Constructors and destructors . . . 120
15.2.1 Do not throw an exception from a destructor . . . 120
15.3 Handling an exception . . . 121
15.3.1 Do not access non-static members from a catch handler of constructor/destructor *function try block* . . . 121
15.3.2 Ensure that a program does not result in a call to std::terminate . . . 122

16 Preprocessing **124**
16.1 Source file inclusion . . . 124
16.1.1 Use the preprocessor only for implementing include guards, and including header files with include guards . . . 124
16.1.2 Do not include a path specifier in filenames supplied in #include directives . . . 125
16.1.3 Match the filename in a #include directive to the one on the filesystem . . . 125
16.1.4 Use <> brackets for system and standard library headers. Use quotes for all other headers . 126
16.1.5 Include directly the minimum number of headers required for compilation . . . 126

17 Standard library **128**
17.1 General . . . 128
17.1.1 Do not use std::vector<bool> . . . 128
17.2 The C standard library . . . 128
17.2.1 Wrap use of the C Standard Library . . . 128
17.3 General utilities library . . . 129
17.3.1 Do not use std::move on objects declared with const or const & type . . . 129
17.3.2 Use std::forward to forward universal references . . . 129

17.3.3 Do not subsequently use the argument to std::forward . . . 130
17.3.4 Do not create smart pointers of array type . . . 130
17.3.5 Do not create an *rvalue reference* of std::array . . . 131
17.4 Containers library . . . 131
17.4.1 Use const container calls when result is immediately converted to a const iterator . . . 131
17.4.2 Use API calls that construct objects in place . . . 132
17.5 Algorithms Library . . . 133
17.5.1 Do not ignore the result of std::remove, std::remove_if or std::unique . . . 133

18 Concurrency 134
18.1 General . . . 134
18.1.1 Do not use platform specific multi-threading facilities . . . 134
18.2 Threads . . . 134
18.2.1 Use high_integrity::thread in place of std::thread . . . 134
18.2.2 Synchronize access to data shared between threads using a single lock . . . 136
18.2.3 Do not share volatile data between threads . . . 138
18.2.4 Use std::call_once rather than the Double-Checked Locking pattern . . . 140
18.3 Mutual Exclusion . . . 141
18.3.1 Within the scope of a lock, ensure that no static path results in a lock of the same mutex . . . 141
18.3.2 Ensure that order of nesting of locks in a project forms a DAG . . . 142
18.3.3 Do not use std::recursive_mutex . . . 143
18.3.4 Only use std::unique_lock when std::lock_guard cannot be used . . . 145
18.3.5 Do not access the members of std::mutex directly . . . 145
18.3.6 Do not use relaxed atomics . . . 145
18.4 Condition Variables . . . 146
18.4.1 Do not use std::condition_variable_any on a std::mutex . . . 146

References 148

High Integrity 3.3 to 4.0 Rule Mappings 149

Revision History 154

Conditions of Use 154

0 Introduction

This document defines a set of rules for the production of high quality C++ code[1]. The guiding principles of this standard are maintenance, portability, readability and robustness. Justification with examples of compliant and/or non-compliant code are provided for each rule. Each rule shall be enforced unless a formal deviation is recorded (see Section 0.6).

This standard adopts the view that restrictions should be placed on the use of the ISO C++ language (see 1.1.1) without sacrificing its core flexibility. This approach allows for the creation of robust and easy to maintain programs while minimizing problems created either by compiler diversity, different programming styles, or dangerous/confusing aspects of the language.

Without applying good coding standards, programmers may write code that is prone to bugs or difficult for someone else to pick up and maintain.

A combination of techniques has to be applied to achieve high integrity software: e.g. requirements management and coverage testing. Only a few of such techniques are programming language specific, of which language subsetting is widely regarded as an effective and scalable method. When rigorously enforced, and coupled with language agnostic techniques, it can facilitate production of high integrity C++ code.

0.1 Typographical Conventions

Throughout this document, a rule is formatted using the following structure.

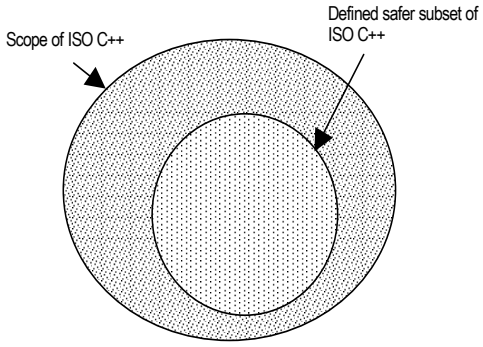
X.Y.Z	This statement describes a rule for C++. Adherence is mandatory. Text immediately below the rule heading provides rationale and example(s).
Exception:	This paragraph explains cases where the rule does not apply.
References:	This section lists sources of relevant material or related rules.
QA·C++ 3.1 Enforcement:	This section details automated enforcement of the rule.
`keyword`	C++ keywords and code items are shown in monospace font and blue color.
term	Terms defined in the C++ Standard appear italicized, see Section 0.7.

0.2 Escalation policy

This coding standard aims to enforce current best practice in C++ development by applying semantic and stylistic recommendations, including controlling the use of language features of C++ which can lead to misunderstanding or errors. In each case a justification is presented as to why the restriction is being applied. However, in view of the fact that research into usage of languages in general and C++ in particular is ongoing, this coding standard will be reviewed and updated from time to time to reflect current best practice in developing reliable C++ code.

[1] Rules have been selected based on their applicability to any project written using the C++ Language. Users of this standard may need to consider additional domain specific rules appropriate to their project.

0.3 Base Standard and Policy



0.3.1 ISO Standard C++

The Base Standard for this document is as specified in 1.1.1 with no extensions allowed and further restrictions as detailed in the rules.

0.3.2 Statically detectable restrictions

This coding standard requires that the use of the C++ language shall be further restricted, so that no reliance on statically detectable[2] undefined or unspecified behavior listed in the ISO C++ Standard is allowed. Where undefined behavior can be identified statically, coding rules limit the potential for introducing it. The rules also prohibit practice which, although well defined, is known to cause problems.

0.3.3 Examples

This standard contains many example code fragments which are designed to illustrate the meaning of the rules. For brevity some of the example code does not conform to all best practices, e.g. unless the rule relates explicitly to concurrency, the example code may not be thread-safe.

0.4 Basis of requirements

Requirements in this standard express restrictions on the use of language constructs or library functions that:

- are not completely defined by the ISO C++ Standard.
- permit varied compiler interpretation.
- are known to be frequently misunderstood or misused by programmers thereby leading to errors.
- do not follow established best practice.

The basis of these requirements is that by meeting them it is possible to avoid known problems and thereby reduce the incidence of errors.

0.5 Rule Enforcement

For any non-trivial code base, manual enforcement of rules in this coding standard will be time consuming and unreliable. Therefore, automated enforcement should be used when possible.

[2] i.e., at compile time

Rules in this coding standard that constrain the value of an expression are undecidable in the computer theoretical sense, because compliance is in general dependent on variables (program state). These rules are identified by the word 'demonstrably' appearing in their headline. However, a program can be augmented with code guards (e.g. assertion statements), to make enforcement decidable, by constraining the value of the expression to guarantee compliance. Conversely, if the value of the expression is not suitably guarded, the code is non-compliant. For an example, see Rule 4.2.2: "Ensure that data loss does not demonstrably occur in an integral expression".

0.6 Deviations

Notwithstanding the requirements of this coding standard, it may be necessary, and in some cases desirable, to tolerate limited non-compliance. Such non-compliance shall, without exception, be the subject of a written deviation supported by a written justification.

0.7 Glossary

term	C++11 ref	explanation
captured by copy	5.1.2	entity captured by a lambda implicitly or explicitly by copy
captured by reference	5.1.2	entity captured by a lambda implicitly or explicitly by reference
function try block	15	a function body which is a try block
lambda-declarator	5.1.2	analogue of function prototype for lambdas
lvalue	3.10	a function or a non-temporary object
one definition rule (ODR)	3.2	places restrictions on multiple definitions of variables, functions, templates and user defined types
ODR use	3.2	for a const object occurs when the object is used as an lvalue, e.g. as an operand of the unary `&` (address of) operator
rvalue	3.10	a temporary object or a value not associated with an object
lvalue reference	8.3.2, 8.5.3	a reference type declared using `&`
rvalue reference	8.3.2, 8.5.3	a reference type declared using `&&`
sequenced before	1.9	establishes a partial order on evaluations executed by a single thread
static initialization	3.6.2	zero-initialization or constant initialization of an object with static or thread storage duration
using declaration	7.3.3	introduces the specified name into the current scope, e.g. `using std::vector;`
using directive	7.3.4	allows all names from the specified namespace to be used in the current scope, e.g. `using namespace std;`

1 General

1.1 Implementation compliance

1.1.1 Ensure that code complies with the 2011 ISO C++ Language Standard

The current version of the C++ language is as defined by the ISO International Standard ISO/IEC 14882:2011(E) ”Information technology – Programming languages – C++”

Compilers often provide features beyond those defined in the Standard, and unrestricted usage of such features will likely hamper code portability. To this end, source code should be routinely parsed with a separate compiler or code analysis tool apart from the compiler used for production purposes.

For Example:

```
#include <cstdint>

void foo (int32_t i)
{
  int32_t * a;

  __try       // Non-Compliant
  {
    a = new int32_t [i];

    // ...
  }
  __finally  // Non-Compliant
  {
    delete [] a;
  }
}
```

References:

- HIC++ v3.3 – 1.3.1
- HIC++ v3.3 – 6.4
- HIC++ v3.3 – 13.3
- Meyers Notes – Reference Binding Rules

1.2 Redundancy

1.2.1 Ensure that all statements are reachable

For the purposes of this rule missing `else` and `default` clauses are considered also.

If a statement cannot be reached for any combination of function inputs (e.g. function arguments, global variables, volatile objects), it can be eliminated.

For example, when the condition of an if statement is never false the `else` clause is unreachable. The entire `if` statement can be replaced with its ’true’ sub-statement only.

In practice two methods are used to detect unreachable code:

- sparse conditional constant propagation
- theorem proving

by showing that non-execution of a statement is independent of function inputs. Since the values of variables are not used to determine unreachability, this restricted definition is decidable (see Section 0.5).

A compiler may detect and silently remove unreachable statements as part of its optimizations. However, explicitly removing unreachable code has other benefits apart from efficiency: the structure of the function will be simplified. This will improve its maintainability and will increase the proportion of statements that can be reached through coverage analysis.

For Example:

```
#include <cstdint>

bool foo (int32_t a)
{
  // ...

  return true;
}

void bar (int32_t b)
{
  // Non-Compliant:  implicit else clause cannot be reached for any 'b'
  if (foo (b))
  {
    // ...
  }

  foo (b); // Compliant
  // ...
}
```

References:

- HIC++ v3.3 – 5.3
- JSF AV C++ Rev C – 186
- MISRA C++:2008 – 0-1-1

1.2.2 Ensure that no expression or sub-expression is redundant

An expression statement with no side effects can be removed or replaced with a null statement without affecting behavior of the program.

Similarly, it is sometimes possible to simplify an expression by removing operands that do not change the resulting value of the expression, for example a multiplication by 1 or 0.

Redundant code causes unnecessary maintenance overhead and may be a symptom of a poor design.

For Example:

```
#include <cstdint>
```

```
void foo (int32_t & a)
{
  a == 0; // Non-Compliant:  was this supposed to be an assignment
}
```

References:

- HIC++ v3.3 – 10.10

1.3 Deprecated features

1.3.1 Do not use the increment operator (++) on a variable of type bool

Incrementing an object with bool type results in its value been set to `true`. This feature was deprecated in the 1998 C++ Language Standard and thus may be withdrawn in a later version.

Prefer to use an explicit assignment to `true`.

For Example:

```
void foo (bool b)
{
  ++b;       // Non-Compliant
}

void bar (bool b)
{
  b = true; // Compliant:  this is equivalent
}
```

References:

- HIC++ v3.3 – 10.16
- C++98 – 5.3.2/1

1.3.2 Do not use the register keyword

Most compilers ignore the `register` keyword, and perform their own register assignments. Moreover, this feature was deprecated in the 2011 C++ Language Standard and thus may be withdrawn in a later version.

For Example:

```
#include <cstdint>

int32_t square (register int32_t a) // Non-Compliant
{
  return a * a;
}
```

References:

- HIC++ v3.3 – 8.3.3
- C++11 – D.2

1.3.3 Do not use the C Standard Library .h headers

The C standard library headers are included in C++ for compatibility. However, their inclusion was deprecated in the 1998 C++ Language Standard and thus they may be withdrawn in a later version.

Instead of <name.h> prefer to use <cname>.

For Example:

```
#include <cstdint>
#include <string.h> // Non-Compliant
#include <cstring>  // Compliant

int32_t foo (const char * s)
{
  return 2 * std::strlen (s);
}
```

References:

- HIC++ v3.3 – 17.1
- C++98 – 17.4.1.2/7

1.3.4 Do not use deprecated STL library features

The following STL features were deprecated in the 2011 C++ Language Standard and thus they may be withdrawn in a later version.

- `std::auto_ptr`
- `std::bind1st`
- `std::bind2nd`
- `std::ptr_mem_fun`
- `std::ptr_mem_fun_ref`
- `std::unary_function`
- `std::binary_function`

Of particular note is `std::auto_ptr` as it has been suggested that a search and replace of this type to `std::unique_ptr`, may uncover latent bugs due to the incorrect use of `std::auto_ptr`.

For Example:

```
#include <cstdint>
#include <memory>

void foo ()
{
  std::auto_ptr<int32_t> p1 (new int32_t(0));    // Non-Compliant
  std::unique_ptr<int32_t> p2 (new int32_t(0)); // Compliant
}
```

References:

- HIC++ v3.3 – 17.21
- Sutter Guru of the Week (GOTW) – 89
- C++11 – D

1.3.5 Do not use `throw` exception specifications

Specifying an exception specification using `throw( type-id-list`$_{opt}$ `)` has been deprecated in the 2011 C++ Language Standard and thus may be removed in a future version.

The new syntax is to use `noexcept` or `noexcept( expr )`.

For Example:

```
#include <cstdint>

void f1 () throw();          // Non-Compliant
void f2 () throw(int32_t);   // Non-Compliant
void f3 () noexcept;         // Compliant
void f4 () noexcept(true);   // Compliant
```

References:

- C++11 – 15.4/17

2 Lexical conventions

2.1 Character sets

2.1.1 Do not use tab characters in source files

Tab width is not consistent across all editors or tools. Code indentation can be especially confusing when tabs and spaces are used interchangeably. This may easily happen where code is maintained using different editors.

In string and character literals \t should be used in preference to a direct tab character.

For Example:

```
#include <cstdint>

void do_something();
void do_something_else();

void foo (int32_t i)
{
    if (i)
        do_something ();
    do_something_else (); // Non-Compliant:  tab character used to indent this statement

}

void bar (int32_t i)
{
    if (i)
        do_something ();
        do_something_else (); // this is what the code looks like with
                              // tab width of 8 instead of 4 as above
}
```

Indenting code only with spaces ensures that formatting is preserved when printing and across different editors or tools.

References:

- HIC++ v3.3 – 14.2

2.2 Trigraph sequences

2.2.1 Do not use digraphs or trigraphs

Trigraphs are special three character sequences, beginning with two question marks and followed by one other character. They are translated into specific single characters, e.g. \ or ^. Digraphs are special two character sequences that are similarly translated.

Be aware of trigraph and digraph character sequences and avoid them. It is possible to avoid such sequences arising accidentally by using spaces.

In most modern environments there is no longer a need for trigraphs or digraphs, and their use will either result in hard to diagnose compiler errors or code that is difficult to maintain.

Trigraph	Equivalent	Digraph	Equivalent
??=	#	%:%:	##
??(	[	%:	#
??<	{	<:	[
??)	]	<%	{
??>	}	:>	]
??/	\	%>	}
??'	^		
??!	|		
??-	~		

For Example:

```
#include <cstdint>
#include <iostream>
#include <vector>

void f1 ()
{
  // Non-Compliant:  here the ??/??/??  becomes \\??  after trigraph translation
  //
  std::cout << "Enter␣date␣??/??/??";
}

void f2 ()
{
  // Non-Compliant:  here the <::std::pair becomes [:std::pair
  //
  ::std::vector<::std::pair<int32_t, int32_t> > vector_of_pairs;
}
```

References:

- HIC++ v3.3 – 14.18

2.3 Comments

2.3.1 Do not use the C comment delimiters /* ... */

The scope of C++ comments is clearer: until the end of a logical source line (taking line splicing into account).

Errors can result from nesting of C comments.

For Example:

```
// Non-Compliant example
void foo (bool isLarge, bool isBright)
{
  /* temporarily disable the code
  if (isLarge)
  {
    if (isBright)
```

```
    {
    /* if isLarge && isBright do something special
    */
    }
  }
  */
  // compilation error on unmatched '*/'
}
```

References:

- HIC++ v3.3 – 14.1

2.3.2 Do not comment out code

Source code should always be under version control. Therefore keeping old code in comments is unnecessary. It can make browsing, searching and refactoring the source code more difficult.

For Example:

```
#include <memory>
#include <cstdint>

void foo_v1 ()
{
  int32_t * p = new int32_t;

  // ...

  delete p;
}

// use RAII for p
void foo_v2 ()
{
  std::unique_ptr<int32_t> p (new int32_t ());

  // ...

  // delete p; // Non-Compliant
}
```

In the above example, use of `std::unique_ptr` means that an explicit `delete` is no longer necessary. However, as the code was commented out rather than removed a reviewer will initially have to spend time confirming that `delete` is no longer necessary.

References:

- JSF AV C++ Rev C – 127
- MISRA C++:2008 – 2-7-3

2.4 Identifiers

2.4.1 Ensure that each identifier is distinct from any other visible identifier

Similarity of identifiers impairs readability, can cause confusion and may lead to mistakes.

Names should not differ only in case (foo/Foo) or in use of underscores (foobar/foo_bar). Additionally, certain combinations of characters look very similar:

O(o)	0	
I(i)	I(L)	1
S(s)	5	
Z(z)	2	
n(N)	h	
B(b)	8	
rn(RN)	m	

Identifiers that only differ in the above characters should also be avoided. This rule applies to pairs of identifiers that can be used in the same scope, i.e. one of them is visible with respect to the other.

Note: This rule does not require that an identifier cannot be reused. See Rule 3.1.1: "Do not hide declarations".

For Example:

```
// t1.cc
#include <cstdint>

int32_t he1lo;     // Non-Compliant
int32_t hel1o;     // Non-Compliant

void FOO ()
{
  int32_t world;   // Compliant:  'wor1d' is not visible
}

void BAR ()
{
  int32_t wor1d;   // Compliant:  'world' is not visible
}

// t2.cc
int32_t F00;       // Non-Compliant:  'FOO' is visible
```

References:

- HIC++ v3.3 – 8.3.4
- MISRA C++:2008 – 2-10-1

2.5 Literals

2.5.1 Do not concatenate strings with different encoding prefixes

The C++ Standard permits string literals with the following encoding prefixes: u, U, u8, L. A program that concatenates a pair of string literals with u8 and L prefixes is ill-formed.

The result of the remaining prefix combinations are all implementation defined. For this reason encoding prefixes should not be mixed.

For Example:

```
auto hw (L"hello" u"world"); // Non-Compliant
```

References:

- HIC++ v3.3 – 6.5

2.5.2 Do not use octal constants (other than zero)

Octal literals are specified with a leading digit `0`; therefore, literal 0 is technically an octal constant.

Do not use any other octal literals, as based on unfamiliarity, this could be confusing and error prone.

For Example:

```
#include <cstdint>

uint32_t v1 (10000); // decimal literal
uint32_t v2 (00010); // Non-Compliant: octal literal with value 8
```

References:

- JSF AV C++ Rev C – 149
- MISRA C++:2008 – 2-13-2

2.5.3 Use `nullptr` for the null pointer constant

The 2011 C++ Language Standard introduced the `nullptr` keyword to denote a null pointer constant.

The `NULL` macro and constant expressions with value `0` can be used in both pointer contexts and integral contexts. `nullptr`, however, is only valid for use in pointer contexts and so cannot be unexpectedly used as an integral value.

For Example:

```
#include <cstdint>
#include <cstddef>

void f1(int32_t);       // #1
void f1(int32_t*);      // #2

int32_t * p = 0;        // Non-Compliant: avoid use of 0 as a null pointer constant
int32_t i = 0;          // Compliant: an integral expression is expected
int32_t * t = nullptr; // Compliant: preferred way to specify a null pointer constant

void f2()
{
  f1(NULL);             // Non-Compliant: Calls #1
  f1(nullptr);          // Compliant: Calls #2
}
```

References:

- HIC++ v3.3 – 14.16
- C++11 – 2.14.7

3 Basic concepts

3.1 Scope

3.1.1 Do not hide declarations

Reusing the same identifier for different declarations is confusing and difficult to maintain. If the hiding declaration is later removed or the identifier is renamed, a compilation error may not be generated, as the declaration that was previously hidden will now be found.

While hidden namespace scope identifiers can still be accesses with a fully qualified name, hidden block scope identifiers will not be accessible.

For Example:

```
#include <cstdint>

void foo (int32_t);

int32_t i;

void bar (int32_t max)
{
  for (int32_t i (0); i < max; ++i)   // Non-Compliant
  {
    for (int32_t i (0); i < max; ++i) // Non-Compliant
    {
      // no way to access the outer loop index
      foo (::i); // namespace scope 'i'.
      foo (i);   // innermost declaration of 'i'
    }
  }
}
```

In C++, it is possible for the same identifier to refer to both a type and an object or a function. In this case the object or function will hide the type.

For Example:

```
#include <cstdint>

// valid C++
class C;
int32_t C;    // Non-Compliant:  object C hides type of same name
```

References:

- HIC++ v3.3 – 8.2.1

3.2 Program and linkage

3.2.1 Do not declare functions at block scope

A declaration for a function should be common to its definition, any redeclarations, and any calls to it.

To ensure that the same type is used in all declarations, functions should always be declared at namespace scope (See Rules 7.4.3: "Ensure that an object or a function used from multiple translation units is declared in a single header file" and 7.4.1: "Ensure that any objects, functions or types to be used from a single translation unit are defined in an unnamed namespace in the main source file").

For Example:

```
#include <cstdint>

int32_t bar ()
{
  int32_t foo ();     // Non-Compliant
  return foo ();
}

int32_t foo ()
{
}
```

References:

- JSF AV C++ Rev C – 107
- MISRA C++:2008 – 3-1-2

3.3 Storage duration

3.3.1 Do not use variables with static storage duration

Variables with linkage (and hence static storage duration), commonly referred to as global variables, can be accessed and modified from anywhere in the translation unit if they have internal linkage, and anywhere in the program if they have external linkage. This can lead to uncontrollable relationships between functions and modules.

Additionally, certain aspects of the order of initialization of global variables are unspecified and implementation defined in the C++ Language Standard. This can lead to unpredictable results for global variables that are initialized at run-time (dynamic initialization).

This rule does not prohibit use of a const object with linkage, so long as:

- it is initialized through *static initialization*
- the object is not *ODR used*

For Example:

```
#include <cstdint>

static int32_t foo ();
extern int32_t ga (foo ()); // Non-Compliant
extern int32_t gb (ga);     // Non-Compliant

namespace
{
  int32_t la (0);           // Non-Compliant
```

```
  const int32_t SIZE (100); // Compliant
}
```

The order of initialization of block scope objects with static storage duration is well defined. However, the lifetime of such an object ends at program termination, which may be incompatible with future uses of the code, e.g. as a shared library. It is preferable to use objects with dynamic storage duration to represent program state, allocated from the heap or a memory pool.

For Example:

```
class Application
{
  // ...
};

Application const & theApp()
{
  static Application app; // Non-Compliant
  return app;
}
```

References:

- HIC++ v3.3 – 8.2.2

3.4 Object lifetime

3.4.1 Do not return a reference or a pointer to an automatic variable defined within the function

The lifetime of a variable with automatic storage duration ends on exiting the enclosing block. If a reference or a pointer to such a variable is returned from a function, the lifetime of the variable will have ended before the caller can access it through the returned handle, resulting in undefined behavior.

For Example:

```
class String
{
public:
  String (char *);
  String (const String &);
};

String & fn1 (char * myArg)
{
  String temp (myArg);
  return temp;          // Non-Compliant: temp destroyed here
}

String fn2 (char * myArg)
{
  String temp (myArg);
  return temp;          // Compliant: the caller will get a copy of temp
}
```

References:

- HIC++ v3.3 – 11.7

3.4.2 Do not assign the address of a variable to a pointer with a greater lifetime

The C++ Standard defines 4 kinds of storage duration:

- static
- thread
- automatic
- dynamic

The lifetime of objects with the first 3 kinds of storage duration is fixed, respectively:

- until program termination
- until thread termination
- upon exiting the enclosing block.

Therefore, undefined behavior will likely occur if an address of a variable with automatic storage duration is assigned to a pointer with static or thread storage duration, or one defined in an outer block. Similarly, for a `thread_local` variable aliased to a pointer with static storage duration.

For Example:

```
#include <cstdint>

void foo (bool b)
{
  int32_t * p;
  if (b)
  {
    int32_t c = 0;
    p = &c;     // Non-Compliant
  }
}
```

If using `high_integrity::thread`, then references or pointers with local storage duration should not be passed into threads that have the `high_integrity::DETACH` property.

For Example:

```
#include <cstdint>
#include "high_integrity.h"

using high_integrity::thread;
using high_integrity::ThreadExec;

void bar(int32_t &);
void foo ()
{
  int32_t i;
  thread<ThreadExec::DETACH> t(bar, std::ref(i)); // Non-Compliant:
                                                  // lifetime of 'i' may end
```

```
                                                    // before thread completes
}
```

References:

- JSF AV C++ Rev C – 173
- MISRA C++:2008 – 7-5-2

3.4.3 Use RAII for resources

Objects with non-trivial destructors and automatic storage duration have their destructors called implicitly when they go out of scope. The destructor will be called both for normal control flow and when an exception is thrown.

The same principle does not apply for a raw handle to a resource, e.g. a pointer to allocated memory. By using a manager class, the lifetime of the resource can be correctly controlled, specifically by releasing it in the destructor.

This idiom is known as Resource Acquisition Is Initialization (RAII) and the C++ Language Standard provides RAII wrappers for many resources, such as:

- dynamically allocated memory, e.g. std::unique_ptr
- files, e.g. std::ifstream
- mutexes, e.g. std::lock_guard

For Example:

```
#include <memory>
#include <cstdint>

void foo_v1 ()
{
  int32_t * p = new int32_t; // Non-Compliant

  // ... possibly throwing an exception - resource not freed

  delete p;
}

void foo_v2 ()
{
  std::unique_ptr<int32_t> p (new int32_t ());  // Compliant

  // ... possibly throwing an exception - resource freed

}
```

The following example demonstrates how RAII can also be used to avoid deadlock when an exception is thrown.

For Example:

```
#include <list>
#include <mutex>
#include <cstdint>

class ListWrapper
{
```

```
public:
  void add1(int32_t val)
  {
    // Non-Compliant: 'unlock' not called if exception thrown by 'push_back'
    mut.lock ();
    lst.push_back(val); // May throw an exception
    mut.unlock ();
  }

  void add2(int32_t val)
  {
    // Compliant: Using lock guarantees unlocking, even where an exception is thrown
    std::lock_guard<std::mutex> lock(mut);
    lst.push_back(val); // May throw an exception
  }

  // ...

private:
  std::list<int32_t> lst;
  mutable std::mutex mut;
};
```

Other benefits of using RAII are:

- clear documentation of resource ownership
- pre/post conditions when accessing memory

For Example:

```
#include <cstdint>
#include <cassert>
#include <memory>

int32_t & f1 ()
{
  int32_t * result (new int32_t ());
  return *result; // Non-Compliant
}

std::unique_ptr<int32_t> f2 ()
{
  std::unique_ptr<int32_t> result (new int32_t ());
  return result; // Compliant
}

void f3 ()
{
  std::weak_ptr<int32_t> p1;
  {
    std::shared_ptr<int32_t> p2 (std::make_shared<int32_t> (0));
    p1 = p2;
  } // p2 goes out of scope
```

```
  // can check if pointer is expired
  assert ( ! p1.expired () && "Ensure is still valid" );
  int32_t i = *p1.lock ();
}
```

References:

- HIC++ v3.3 – 3.2.5
- HIC++ v3.3 – 9.5
- HIC++ v3.3 – 12.5
- HIC++ v3.3 – 12.8
- Williams Concurrency – 3.2.1
- CERT C++ – CON02-CPP

3.5 Types

3.5.1 Do not make any assumptions about the internal representation of a value or object

Avoid C++ constructs and practices that are likely to make your code non-portable:

- A union provides a way to alter the type ascribed to a value without changing its representation. This reduces type safety and is usually unnecessary. In general it is possible to create a safe abstraction using polymorphic types.
- Integer types other than signed / unsigned char have implementation defined size. Do not use integer types directly, instead use size specific typedefs, defined in a common header file, which can then be easily adjusted to match a particular platform.
- Do not mix bitwise and arithmetic operations on the same variable, as this is likely to be non portable between big and little endian architectures.
- Do not assume the layout of objects in memory, e.g. by comparing pointers to different objects with the relational operators, using the `offsetof` macro, or performing pointer arithmetic within an object with unspecified or implementation defined layout.

For Example:

```
#include <cstdint>

union U                 // Non-Compliant
{
  float f;
  int32_t i;            // Non-Compliant
};

uint32_t foo (uint32_t u)
{
  --u;
  return u & 0xFFU; // Non-Compliant:  mixing arithmetic and bitwise operations
}

bool cmp (int32_t * lhs, int32_t * rhs)
```

```
{
  return lhs < rhs; // Non-Compliant
}
```

References:

- HIC++ v3.3 – 13.6
- HIC++ v3.3 – 15.1
- JSF AV C++ Rev C – 210
- JSF AV C++ Rev C – 210.1
- JSF AV C++ Rev C – 147
- JSF AV C++ Rev C – 215
- MISRA C++:2008 – 3-9-3
- MISRA C++:2008 – 5-0-15

4 Standard conversions

4.1 Array-to-pointer conversion

4.1.1 Ensure that a function argument does not undergo an array-to-pointer conversion

When an array is bound to a function parameter of pointer type the array is implicitly converted to a pointer to the first element of the array.

In order to retain the array dimension, the parameter should be changed to a reference type or changed to a user defined type such as `std::array`.

For Example:

```
#include <cstdint>
#include <array>

void f (int32_t a[10]); // parameter is of pointer type
void g (int32_t a[]);   // parameter is of pointer type
void h (int32_t * a);
void i (int32_t (&a) [10]);
void j (std::array<int32_t, 10> & a);

void foo ()
{
  int32_t a1 [10];
  std::array<int32_t, 10> a2;

  f(a1); // Non-Compliant
  g(a1); // Non-Compliant
  h(a1); // Non-Compliant
  i(a1); // Compliant
  j(a2); // Compliant
}
```

References:

- JSF AV C++ Rev C – 97
- MISRA C++:2008 – 5-2-12

4.2 Integral conversions

4.2.1 Ensure that the `U` suffix is applied to a literal used in a context requiring an unsigned integral expression

If a literal is used to initialize a variable of unsigned type, or with an operand of unsigned type in a binary operation, the `U` suffix should be appended to the literal to circumvent an implicit conversion, and make the intent explicit.

For Example:

```
#include <cstdint>
```

```
void foo ()
{
  uint32_t u (0);   // Non-Compliant

  u += 2;           // Non-Compliant
  u += 2U;          // Compliant
}
```

References:

- HIC++ v3.3 – 6.1
- MISRA C++:2008 – 2-13-3

4.2.2 Ensure that data loss does not demonstrably occur in an integral expression

Data loss can occur in a number of contexts:

- implicit conversions
- type casts
- shift operations
- overflow in signed arithmetic operations
- wraparound in unsigned arithmetic operations

If possible, integral type conversions should be avoided altogether, by performing all operations in a uniform type matched to the execution environment.

Where data storage is a concern, type conversions should be localized with appropriate guards (e.g. assertions) to detect data loss.

Similar techniques can be used to guard shift and arithmetic operations, especially where the data is tainted in a security sense, i.e. a malicious user can trigger data loss with appropriately crafted input data.

For Example:

```
#include <climits>
#include <stdexcept>
#include <cstdint>

uint32_t inv_mult (uint32_t a, uint32_t b)
{
  return ((0 == a) || (0 == b)) ? UINT_MAX
    : (1000 / (a * b)); // Non-Compliant:  could wraparound
}

void foo ()
{
  inv_mult (0x10000u, 0x10000u);
}

uint32_t safe_inv_mult (uint32_t a, uint32_t b)
{
  if ((b != 0) && (a > (UINT_MAX / b)))
  {
```

```
    throw std::range_error ("overflow");
  }
  return ((0 == a) || (0 == b)) ? UINT_MAX
    : (1000 / (a * b)); // Compliant:  wraparound is not possible
}
```

Data loss may also occur if high order bits are lost in a left shift operation, or the right hand operand of a shift operator is so large that the resulting value always 0 or undefined regardless of the value of the left hand operand.

Therefore, appropriate safeguards should be coded explicitly (or instrumented by a tool) to ensure that data loss does not occur in shift operations.

For Example:

```
#include <cstdint>

void foo (uint8_t u)
{
  uint32_t v = u >> 8U;   // Non-Compliant:  always 0
  v <<= 32U;              // Non-Compliant:  undefined behavior
  v = 0xF1234567U << 1;   // Non-Compliant:  high bit is lost
}
```

For the purposes of this rule integral to `bool` conversions are considered to results in data loss as well. It is preferable to use equality or relational operators to replace such type conversions. The C++ Language Standard states that unless the condition of an `if`, `for`, `while` or `do` statement has boolean type, it will be implicitly converted to `bool`.

For Example:

```
#include <cstdint>

int32_t foo ();

void bar (int32_t i)
{
  if (i)          // Non-Compliant
  {
  }

  if (i != 0)     // Compliant
  {
  }

  for (int32_t j (10); j ; --j) // Non-Compliant
  {
  }

  while (int32_t j = foo ())     // Non-Compliant
  {
  }
}
```

Note: An implicit conversion using an `operator bool` declared as `explicit` does not violate this rule.

References:

- HIC++ v3.3 – 5.2
- HIC++ v3.3 – 8.4.13
- HIC++ v3.3 – 10.7
- HIC++ v3.3 – 10.12
- HIC++ v3.3 – 10.13

4.3 Floating point conversions

4.3.1 Do not convert an expression of wider floating point type to a narrower floating point type

The C++ Standard defines 3 floating point types: `float`, `double`, `long double` that, at least conceptually, increase in precision.

Expressions that implicitly or explicitly cause a conversion from `long double` type to `float` or `double`, and from `double` to `float` should be avoided as they may result in data loss.

When using a literal in a context that requires type `float`, use the `F` suffix, and for consistency use the `L` suffix in a `long double` context.

For Example:

```
void foo ()
{
  float f (1.0);          // Non-Compliant
  f = 1.0F;               // Compliant

  double d (1.0L);        // Non-Compliant
  d = 1.0;                // Compliant

  long double ld (1.0); // Compliant, but not good practice
  ld = 1.0L;              // Compliant

  f = ld;                 // Non-Compliant
  d = ld;                 // Non-Compliant
  f = d;                  // Non-Compliant

  d = f;                  // Compliant
  ld = f;                 // Compliant
  ld = d;                 // Compliant
}
```

References:

- HIC++ v3.3 – 6.2
- HIC++ v3.3 – 10.14

4.4 Floating-integral conversions

4.4.1 Do not convert floating values to integral types except through use of standard library functions

An implicit or explicit conversion from a floating to an integral type can result in data loss due to the significant difference in the respective range of values for each type.

Additionally, floating point to integral type conversions are biased as the fractional part is simply truncated instead of being rounded to the nearest integral value. For this reason use of standard library functions: `std::floor` and `std::ceil` is recommended if a conversion to an integral type is necessary.

For Example:

```
#include <cstdint>
#include <cmath>

void foo (double d)
{
  int32_t i = d;              // Non-Compliant, fraction is truncated
  i = d + 0.5;                // Non-Compliant, number is rounded
  i = std::floor (d);         // Compliant, fraction is truncated
  i = std::floor (d + 0.5); // Compliant, number is rounded
}
```

Note: A return value of `std::floor` and `std::ceil` is of floating type, and an implicit or explicit conversion of this value to an integral type is permitted.

References:

- HIC++ v3.3 – 7.6

5 Expressions

5.1 Primary expressions

5.1.1 Use symbolic names instead of literal values in code

Use of ”magic” numbers and strings in expressions should be avoided in preference to constant variables with meaningful names.

The use of named constants improves both the readability and maintainability of the code.

For Example:

```
#include <iostream>
#include <cstdint>

namespace
{
  const int32_t MAX_ITERATIONS (10);
  const char * const LOOP_ITER_S ("iter␣");
  const char SEP_C (':');
}
void foo ()
{
  for (int32_t i = 0 ; i < 10; ++i)                    // Non-Compliant
  {
    std::cout << "iter␣" << i << ':' << std::endl;     // Non-Compliant
    // ...
  }

  for (int32_t i = 0 ; i < MAX_ITERATIONS; ++i)        // Compliant
  {
    std::cout << LOOP_ITER_S << i << SEP_C << std::endl; // Compliant
    // ...
  }
}
```

See 7.4: ”Linkage specifications” for guidance on where to declare such symbolic names.

References:

- HIC++ v3.3 – 10.1

5.1.2 Do not rely on the sequence of evaluation within an expression

To enable optimizations and parallelization, the C++ Standard uses a notion of *sequenced before*, e.g.:

- evaluation of a full expression is sequenced before the next full-expression to be evaluated
- evaluation of operands of an operator are sequenced before the evaluation of the operator
- evaluation of arguments in a function call are sequenced before the execution of the called function
- for built-in operators `&&`, `||`, `,` and operator `?` evaluation of the first operand is sequenced before evaluation of the other operand(s).

This defines a partial order on evaluations, and where two evaluations are unsequenced with respect to one another, their execution can overlap. Additionally, two evaluations may be indeterminately sequenced, which is similar, except that the execution cannot overlap.

This definition leaves great latitude to a compiler to re-order evaluation of sub-expressions, which can lead to unexpected, and even undefined behavior. For this reason, and to improve code readability an expression should not:

- have more than one side effect
- result in the modification and access of the same scalar object
- include a sub-expression that is an assignment operation
- include a sub-expression that is a pre- or post-increment/decrement operation
- include a built-in comma operator (for overloaded comma operator see Rule 13.2.1: "Do not overload operators with special semantics")

For Example:

```
#include <cstdint>

int32_t foo (int32_t i, int32_t j)
{
  int32_t k = ++i + ++j; // Non-Compliant:  two side effects in full expression
  k = ++i;               // Non-Compliant:  pre-increment as a sub-expression
  ++i;                   // Compliant:  pre-increment as an expression statement
  ++j;                   // Compliant
  k = i + j;             // Compliant

  k = i + ++i;           // Non-Compliant:  undefined behavior

  if (k = 0)             // Non-Compliant:  assignment as a condition expression
  {
  }

  return i, j;           // Non-Compliant:  built-in comma operator used
}
```

References:

- HIC++ v3.3 – 10.3
- HIC++ v3.3 – 10.5
- HIC++ v3.3 – 10.19
- JSF AV C++ Rev C – 204
- MISRA C++:2008 – 5-2-10

5.1.3 Use parentheses in expressions to specify the intent of the expression

The effects of precedence and associativity of operators in a complicated expression can be far from obvious. To enhance readability and maintainability of code, no reliance on precedence or associativity in an expression should be made, by using explicit parentheses, except for

- operands of assignment

- any combination of + and - operations only
- any combination of * and / operations only
- sequence of && operations only
- sequence of || operations only

For Example:

```
#include <cstdint>

int32_t foo (int32_t i, int32_t j)
{
  int32_t k;
  k = i + j;               // Compliant
  int32_t r = i + j * k; // Non-Compliant
  r = i + j + k;           // Compliant

  // Compliant
  if ((i != 0) && (j != 0) && (k != 0))
  {
  }

  // Non-Compliant
  if ((i != 0) && (j != 0) || (k != 0))
  {
  }

  // Compliant
  if (((i != 0) && (j != 0)) || (k != 0))
  {
  }

  return i + j + k;        // Compliant
}
```

References:

- HIC++ v3.3 – 10.4

5.1.4 Do not capture variables implicitly in a lambda

Capturing variables helps document the intention of the author. It also allows for different variables to be *captured by copy* and *captured by reference* within the same lambda.

For Example:

```
#include <cstddef>
#include <vector>
#include <algorithm>
void foo (std::vector<size_t> const & v)
{
  size_t sum = 0;
  std::for_each(v.cbegin ()
    , v.cend ()
```

```
    , [&](size_t s) { sum += s; } );     // Non-Compliant

  sum = 0;
  std::for_each(v.cbegin ()
    , v.cend ()
    , [&sum](size_t s) { sum += s; } ); // Compliant
}
```

Exception:

It is not necessary to capture objects with static storage duration or constants that are not *ODR used*.

However, the use of objects with static storage duration should be avoided. See Rule 3.3.1: "Do not use variables with static storage duration".

For Example:

```
#include <cstddef>
#include <cstdint>

void foo ()
{
  const size_t N = 10;
  static int32_t j = 0; // Non-Compliant:  object with static storage duration
  [](size_t s)
  {
  int32_t array[N];      // Compliant:  Not ODR used
    ++j;                 // Compliant
  };
}
```

5.1.5 Include a (possibly empty) parameter list in every lambda expression

The *lambda-declarator* is optional in a lambda expression and results in a closure that can be called without any parameters.

To avoid any visual ambiguity with other C++ constructs, it is recommended to explicitly include `( )`, even though it is not strictly required.

For Example:

```
#include <cstdint>

int32_t i;
int32_t j;
void foo ()
{
  int32_t a[]    { ++i, ++j  } ;   // Not a lambda
           []    { ++i, ++j ;} ;   // Non-Compliant
           [] () { ++i, ++j ;} ;   // Compliant
}
```

5.1.6 Do not code side effects into the right-hand operands of: `&&`, `||`, `sizeof`, `typeid` or a function passed to `condition_variable::wait`

For some expressions, the side effect of a sub-expression may not be evaluated at all or can be conditionally evaluated, that is, evaluated only when certain conditions are met. For Example:

- The right-hand operands of the `&&` and `||` operators are only evaluated if the left hand operand is `true` and `false` respectively.
- The operand of `sizeof` is never evaluated.
- The operand of `typeid` is evaluated only if it is a function call that returns reference to a polymorphic type.

Having visible side effects that do not take place, or only take place under special circumstances makes the code harder to maintain and can also make it harder to achieve a high level of test coverage.

For Example:

```
#include <typeinfo>

bool doSideAffect();

class C
{
public:
  virtual ~C(); // polymorphic class
};

C& foo();

void foo( bool condition )
{
  if (false && doSideAffect ()) // Non-Compliant:  doSideAffect not called
  {}

  if (true || doSideAffect ())  // Non-Compliant:  doSideAffect not called
  {}

  sizeof (doSideAffect());      // Non-Compliant:  doSideAffect not called
  typeid (doSideAffect());      // Non-Compliant:  doSideAffect not called
  typeid (foo());               // Non-Compliant:  foo called to determine
                                // the polymorphic type
}
```

Conditional Variable: wait member

Every time a waiting thread wakes up, it checks the condition. The wake-up may not necessarily happen in direct response to a notification from another thread. This is called a spurious wake. It is indeterminate how many times and when such spurious wakes happen. Therefore it is advisable to avoid using a function with side effects to perform the condition check.

For Example:

```
#include <mutex>
#include <condition_variable>
#include <cstdint>


std::mutex mut;
std::condition_variable cv;
```

```
int32_t i;

bool sideEffects()
{
  ++i;
  return (i > 10);
}

void threadX()
{
  i = 0;
  std::unique_lock<std::mutex> guard(mut);
  cv.wait(guard, sideEffects);  // Non-Compliant
                                // value of i depends on the number of wakes
}
```

References:

- HIC++ v3.3 – 10.9

5.2 Postfix expressions

5.2.1 Ensure that pointer or array access is demonstrably within bounds of a valid object

Unlike standard library containers, arrays do not benefit from bounds checking.

Array access can take one of the equivalent forms: `*(p + i)` or `p[i]`, and will result in undefined behavior, unless `p` and `p + i` point to elements of the same array object. Calculating (but not dereferencing) an address one past the last element of the array is well defined also. Note that a scalar object can be considered as equivalent to an array dimensioned to 1.

To avoid undefined behavior, appropriate safeguards should be coded explicitly (or instrumented by a tool), to ensure that array access is within bounds, and that indirection operations (`*`) will not result in a null pointer dereference.

For Example:

```
#include <cassert>
#include <cstdint>


void foo (int32_t* i)
{
  int32_t k = *i;            // Non-Compliant:  foo could be called with a null pointer

  assert (i != nullptr);
  k = *i;                    // Compliant

  int32_t a [10];
  for (int32_t i (0); i < 10; ++i)
  {
    a [i] = i;               // Compliant, array index is 0..9
  }
```

```
  int32_t * p = & (a [10]); // Compliant:  calculating one past the end of array
  k = *p;                   // Non-Compliant:  out of bounds access
}
```

References:

- HIC++ v3.3 – 10.2

5.2.2 Ensure that functions do not call themselves, either directly or indirectly

As the program stack tends to be one of the more limited resources, excessive use of recursion may limit the scalability and portability of the program. Tail recursion can readily be replaced with a loop. Other forms of recursion can be replaced with an iterative algorithm and worklists.

For Example:

```
#include <cstdint>

int32_t a (int32_t);
int32_t b (int32_t);
int32_t c (int32_t);
int32_t d (int32_t);
int32_t e (int32_t);
int32_t f (int32_t);
int32_t g (int32_t);
int32_t h (int32_t);

int32_t foo (int32_t v)
{
  if (a (v))
  {
    return e (v);
  }
  else if (b (v))
  {
    return foo (f (v)); // Non-Compliant:  tail recursion
  }
  else if (c (v))
  {
    return g (v);
  }
  else if (d (v))
  {
    return foo (h (v)); // Non-Compliant:  tail recursion
  }
}

// Compliant:  equivalent algorithm
int32_t bar (int32_t v)
{
  for (;;)
  {
    if (a (v))
```

```
    {
      v = e (v);
      break;
    }
    else if (b (v))
    {
      v = f (v);
    }
    else if (c (v))
    {
      v = g (v);
      break;
    }
    else if (d (v))
    {
      v = h (v);
    }
  }
  return v;
}
```

References:

- JSF AV C++ Rev C – 119
- MISRA C++:2008 – 7-5-4

5.3 Unary expressions

5.3.1 Do not apply unary minus to operands of unsigned type

The result of applying a unary minus operator (-) to an operand of unsigned type (after integral promotion) is a value that is unsigned and typically very large.

Prefer to use the bitwise complement (~) operator instead.

For Example:

```
#include <cstdint>

void foo ()
{
  uint32_t v;
  v = -1u; // Non-Compliant
  v = ~0u; // Compliant
}
```

References:

- HIC++ v3.3 – 10.21

5.3.2 Allocate memory using `new` and release it using `delete`

C style allocation is not type safe, and does not invoke constructors or destructors. For this reason only operators `new` and `delete` should be used to manage objects with dynamic storage duration.

Note: Invoking `delete` on a pointer allocated with `malloc` or invoking `free` on a pointer allocated with `new` will result in undefined behavior.

For Example:

```
#include <cstdlib>
#include <cstdint>


void foo ()
{
  // Non-Compliant
  int32_t * i = static_cast <int32_t *> (std::malloc (sizeof (int32_t)));
  std::free (i);
}

void bar ()
{
  // Compliant
  int32_t * i = new int32_t;
  delete i;
}
```

References:

- 3.4.3: Use RAII for resources
- HIC++ v3.3 – 12.2

5.3.3 Ensure that the form of `delete` matches the form of `new` used to allocate the memory

The C++ Standard requires that the operand to the `delete` operator is either:

- a null pointer
- pointer to a non array object allocated with `new`
- pointer to a base class[3] subobject of a non array object allocated with `new`

Similarly, the operand to the `delete[]` operator is either:

- a null pointer
- pointer to an array object allocated with `new[]`

In order to avoid undefined behavior, plain and array forms of `delete` and `new` should not be mixed.

For Example:

```
#include <cstdint>

void foo ()
{
  int32_t * i = new int32_t [10];
```

[3] containing a virtual destructor

```
  // ...

  delete i; // Non-Compliant
}

typedef int32_t ARRAY [10];

void bar ()
{
  int32_t * i = new ARRAY;

  delete i; // Non-Compliant
}
```

References:

- HIC++ v3.3 – 12.3

5.4 Explicit type conversion

5.4.1 Only use casting forms: `static_cast` (excl. `void*`), `dynamic_cast` or explicit constructor call

All casts result in some degree of type punning, however, some casts may be considered more error prone than others:

- It is undefined behavior for the result of a `static_cast` to `void*` to be cast to any type other than the original from type.
- Depending on the type of an object, casting away `const` or `volatile` and attempting to write to the result is undefined behavior.
- Casts using `reinterpret_cast` are generally unspecified and/or implementation defined. Use of this cast increases the effort required to reason about the code and reduces its portability.
- Simplistically, a C-style cast and a non class function style cast can be considered as a sequence of the other cast kinds. Therefore, these casts suffer from the same set of problems. In addition, without a unique syntax, searching for such casts in code is extremely difficult.

For Example:

```
#include <cstdint>

void bar (int8_t);

void foo (const int32_t * p)
{
  int32_t * r = const_cast <int32_t *> (p);    // Non-Compliant: casting away const

  int32_t i = reinterpret_cast <int32_t> (r); // Non-Compliant

  i = (int32_t) r;                             // Non-Compliant: C-style cast

  bar (int8_t (i));                            // Non-Compliant: function style cast
```

```
  bar (static_cast <int8_t> (i));              // Compliant
}

class Base
{
public:
  virtual ~Base ();
};

class Derived : virtual public Base
{
public:
  Derived (int32_t);
};

void foo (Base * base)
{
  Derived * d;
  d = reinterpret_cast<Derived *> (base); // Non-Compliant
  d = dynamic_cast<Derived *> (base);     // Compliant
  auto d2 = Derived (0);                  // Compliant:  Explicit constructor call
}
```

References:

- 5.4.3: Do not convert from a base class to a derived class
- HIC++ v3.3 – 7.1
- HIC++ v3.3 – 7.3
- HIC++ v3.3 – 7.4
- HIC++ v3.3 – 7.5
- HIC++ v3.3 – 7.7
- HIC++ v3.3 – 13.7
- MISRA C++:2008 – 5-2-2
- MISRA C++:2008 – 5-2-7
- MISRA C++:2008 – 5-2-8
- MISRA C++:2008 – 5-2-9

5.4.2 Do not cast an expression to an enumeration type

The result of casting an integer to an enumeration type is unspecified if the value is not within the range of the enumeration. This also applies when casting between different enumeration types.

For this reason conversions to an enumeration type should be avoided.

For Example:

```
enum Colors { RED, GREEN = 2, BLUE };
void bar()
{
```

```
  Colors color = static_cast <Colors> (1000); // Non-Compliant:
  if (1000 == color)                          // may be false
  {
  }
}

void foo()
{
  Colors color = static_cast <Colors> (1);    // Non-Compliant
  switch (color)                              // the value is unspecified
  {
  case RED:
  case GREEN:
  case BLUE:
    break;
  default:
    break;
  }
}
```

References:

- HIC++ v3.3 – 15.4

5.4.3 Do not convert from a base class to a derived class

The most common reason for casting down an inheritance hierarchy, is to call derived class methods on an object that is a reference or pointer to the base class.

Using a virtual function removes the need for the cast completely and improves the maintainability of the code.

For Example:

```
class A
{
public:
  virtual void bar();
};

class B : public A
{
public:
  void bar() override;
  void foo();
};

void foo (A* a)
{
  (dynamic_cast<B*> (a))->foo(); // Non-Compliant
  a->bar();                      // Compliant
}
```

Where the cast is unavoidable, `dynamic_cast` should be used in preference to `static_cast` as the compiler will check the validity of the cast at runtime.

References:

- 5.4.1: Only use casting forms: `static_cast` (excl. `void*`), `dynamic_cast` or explicit constructor call
- HIC++ v3.3 – 3.3.3

5.5 Multiplicative operators

5.5.1 Ensure that the right hand operand of the division or remainder operators is demonstrably non-zero

The result of integer division or remainder operation is undefined if the right hand operand is zero. Therefore, appropriate safeguards should be coded explicitly (or instrumented by a tool) to ensure that division by zero does not occur.

For Example:

```
#include <cstdint>
#include <cassert>

int32_t doDivide(int32_t number, int32_t divisor)
{
  assert (0 != divisor);
  return number / divisor;
}
```

References:

- HIC++ v3.3 – 10.17

5.6 Shift operators

5.6.1 Do not use bitwise operators with signed operands

Use of signed operands with bitwise operators is in some cases subject to undefined or implementation defined behavior. Therefore, bitwise operators should only be used with operands of unsigned integral types.

For Example:

```
#include <cstdint>

void foo (int32_t i)
{
  int32_t r = i << -1;     // Non-Compliant:  undefined behavior
  r = -1 >> 1;             // Non-Compliant:  implementation defined
  r = ~0;                  // Non-Compliant:  implementation defined

  uint32_t u = (-1) & 2u; // Non-Compliant:  implementation defined
  u = (-1) | 1u;           // Non-Compliant:  implementation defined
  u = (-1) ^ 1u;           // Non-Compliant:  implementation defined
}
```

References:

- HIC++ v3.3 – 10.11

5.7 Equality operators

5.7.1 Do not write code that expects floating point calculations to yield exact results

Floating point calculations suffer from machine precision (epsilon), such that the exact result may not be representable. Epsilon is defined as the difference between 1 and the smallest value greater than 1 that can be represented in a given floating point type. Therefore, comparisons of floating point values need to take epsilon into account.

For Example:

```
#include <cmath>
#include <limits>

bool isEqual( const double a, const double b )
{
  const double scale = ( std::fabs( a ) + std::fabs( b ) ) / 2.0;

  return std::fabs( a - b ) <= ( std::numeric_limits<double>::epsilon()
                                 * scale );
}

void foo( double f )
{
  if (3.142 == f)            // Non-Compliant
  {}

  if (isEqual (f, 3.142))    // Compliant
  {}
}
```

References:

- HIC++ v3.3 – 10.15

5.7.2 Ensure that a pointer to member that is a virtual function is only compared (==) with nullptr

The result of comparing a pointer to member to a virtual function to anything other than nullptr is unspecified.

For Example:

```
class A
{
public:
  void f1();
  void f2();
  virtual void f3();
};

void foo( )
{
  if (&A::f1 == &A::f2);   // Compliant
  if (&A::f1 == nullptr); // Compliant
  if (&A::f3 == &A::f2);   // Not Compliant
```

```
  if (&A::f3 == nullptr); // Compliant
}
```

References:

- JSF AV C++ Rev C – 97.1

5.8 Conditional operator

5.8.1 Do not use the conditional operator (?:) as a sub-expression

Evaluation of a complex condition is best achieved through explicit conditional statements (if/else). Using the result of the conditional operator as an operand reduces the maintainability of the code.

The only permissible uses of a conditional expression are:

- argument expression in a function call
- return expression
- initializer in a member initialization list
- object initializer
- the right hand side operand of assignment (excluding compound assignment)

The last use is allowed on the basis of initialization of an object with automatic storage duration being equivalent to its declaration, followed by assignment.

For Example:

```
#include <cstdint>

void foo (int32_t i, int32_t j)
{
  int32_t k;
  k = (j != 0) ? 1 : 0;        // Compliant: equivalent to initialization

  // Non-Compliant: nested conditional operations
  k = (i != 0) ? ((j != 0) ? 1 : 0) : 0;

  k = i + ((j != 0) ? 1 : 0); // Non-Compliant
}
```

References:

- HIC++ v3.3 – 10.20

6 Statements

6.1 Selection statements

6.1.1 Enclose the body of a selection or an iteration statement in a compound statement

Follow each control flow primitive (`if`, `else`, `while`, `for`, `do` and `switch`) by a block enclosed by braces, even if the block is empty or contains only one line. Use of null statements or statement expressions in these contexts reduces code readability and making it harder to maintain.

For Example:

```
#include <cstdint>

void doSomething ();

void foo (int32_t i)
{
  if (0 == i)
    doSomething (); // Non-Compliant
  else
    ;               // Non-Compliant

  if (0 == i)
  {                 // Compliant
    doSomething ();
  }
  else
  {                 // Compliant
  }

  switch (i)
  case 0:
    doSomething (); // Non-Compliant
}
```

References:

- HIC++ v3.3 – 5.1

6.1.2 Explicitly cover all paths through multi-way selection statements

Make sure that each if-else-if chain has a final else clause, and every switch statement has a default clause. The advantage is that all execution paths are explicitly considered, which in turn helps reduce the risk that an unexpected value will result in incorrect execution.

For Example:

```
#include <cstdint>

void foo (int32_t i)
{
  // Non-Compliant:  missing else clause to cover 0 == i path
```

```
  if (i > 0)
  {
  }
  else if (i < 0)
  {
  }

  // Non-Compliant:  missing default clause
  switch (i)
  {
  case 0:
    break;
  case 1:
    break;
  }
}
```

References:

- HIC++ v3.3 – 5.11

6.1.3 Ensure that a non-empty case statement block does not fall through to the next label

Fall through from a non empty case block of a switch statement makes it more difficult to reason about the code, and therefore harder to maintain.

For Example:

```
#include <cstdint>

void foo (int32_t i)
{
  switch (i)
  {
  case 0:       // Compliant
  case 1:
    ++i;
    break;
  case 2:       // Non-Compliant
    ++i;
  default:
    break;
  }
}
```

References:

- HIC++ v3.3 – 5.4

6.1.4 Ensure that a switch statement has at least two case labels, distinct from the default label

A switch statement with fewer than two case labels can be more naturally expressed as a single if statement.

For Example:

```
#include <cstdint>

void doSomething ();
void doSomethingElse ();

void foo (int32_t i)
{
  // Non-Compliant:  1 case label
  switch (i)
  {
  case 0:
    doSomething ();
    break;
  default:
    doSomethingElse ();
    break;
  }

  // Compliant:  an equivalent if statement
  if (0 == i)
  {
    doSomething ();
  }
  else
  {
    doSomethingElse ();
  }

  // Non-Compliant:  only 1 case label distinct from the default label
  switch (i)
  {
  case 0:
    doSomething ();
    break;
  case 1:
  default:
    doSomethingElse ();
    break;
  }

  // Compliant:  2 case labels distinct from the default label
  switch (i)
  {
  case 0:
  case 1:
    doSomething ();
    break;
  default:
    doSomethingElse ();
    break;
  }
}
```

Note: By virtue of this rule and Rule 1.2.1: "Ensure that all statements are reachable", switch statements with a boolean condition should not be used.

For Example:

```
void bar (bool b)
{
  switch (b)
  {
  case true:
    break;
  case false:
    break;
  default:
    break;  // Non-Compliant: unreachable statement
  }

  switch (b)
  {
  case true:
    break;
  case false:
  default:
    break;  // Non-Compliant: only 1 case label distinct from the default label
  }
}
```

References:

- JSF AV C++ Rev C – 195
- JSF AV C++ Rev C – 196
- MISRA C++:2008 – 6-4-7
- MISRA C++:2008 – 6-4-8

6.2 Iteration statements

6.2.1 Implement a loop that only uses element values as a range-based loop

A range-based for statement reduces the amount of boilerplate code required to maintain correct loop semantics.

A range-based loop can normally replace an explicit loop where the index or iterator is only used for accessing the container value.

For Example:

```
#include <iterator>
#include <cstdint>

void bar ()
{
  uint32_t array[] = { 0, 1, 2, 3, 4, 5, 6 };
  uint32_t sum = 0;
```

```
  // Non-Compliant
  for (uint32_t * p = std::begin (array); p != std::end (array); ++p)
  {
    sum += *p;
  }

  sum = 0;
  // Compliant
  for (uint32_t v : array )
  {
    sum += v;
  }

  // Compliant
  for (size_t i = 0; i != (sizeof(array)/sizeof(*array)); ++i)
  {
    if ((i % 2) == 0)        // Using the loop index
    {
      sum += array[i];
    }
  }

}
```

6.2.2 Ensure that a loop has a single loop counter, an optional control variable, and is not degenerate

A loop is considered 'degenerate' if:

- when entered, the loop is infinite, or
- the loop will always terminate after the first iteration.

To improve maintainability it is recommended to avoid degenerate loops and to limit them to a single counter variable.

For Example:

```
#include <cstdint>

int32_t foo ();

void bar (int32_t max)
{
  // Non-Compliant:  2 loop counters
  for (int32_t i (0), j (foo ()) ; (i < max) && (j > 0); ++i, j = foo ())
  {
  }

  bool keepGoing (true);
  // Compliant:  1 loop counter and 1 control variable
  for (int32_t i (0) ; keepGoing && (i < max); ++i)
  {
    keepGoing = foo () > 0;
  }
```

```
  for (int32_t i (0) ; i < max; ++i) // Compliant
  {
    if (foo () <= 0)
    {
      break;
    }
  }
}
```

References:

- MISRA C++:2008 – 6-5-1
- MISRA C++:2008 – 6-5-6

6.2.3 Do not alter a control or counter variable more than once in a loop

The behavior of iteration statements with multiple modifications of control or counter variables is difficult to understand and maintain.

For Example:

```
#include <cstdint>

void foo()
{
  for ( int32_t i (0); i != 10; ++i ) //Non-Compliant:  does this loop terminate?
  {
    if ( 0 == i % 3 )
    {
      ++i;
    }
  }
}
```

References:

- HIC++ v3.3 – 5.6

6.2.4 Only modify a for loop counter in the for expression

It is expected that a for loop counter is modified for every iteration. To improve code readability and maintainability, the counter variable should be modified in the loop expression.

For Example:

```
#include <cstdint>

bool foo ();

void bar (int32_t max)
{
  for (int i (0) ; i < max; ) // Non-Compliant
  {
```

```
    if (foo ())
    {
      ++i;
    }
  }
}
```

References:

- HIC++ v3.3 – 5.5

6.3 Jump statements

6.3.1 Ensure that the label(s) for a jump statement or a switch condition appear later, in the same or an enclosing block

Backward jumps and jumps into nested blocks make it more difficult to reason about the flow through the function.

Loops should be the only constructs that perform backward jumps, and the only acceptable use of a goto statement is to jump forward to an enclosing block.

For Example:

```
#include <cstdint>

void f1 (int32_t i)
{
start:            // Non-Compliant
  ++i;
  if (i < 10)
  {
    goto start;
  }
}

void f2 (int32_t i)
{
  do              // Compliant:  Same as 'f1' using do/while loop
  {
    ++i;
  } while (i < 10);
}


bool f3 (int (&array)[10][10])
{
  for (int j1 = 0; j1 < 10; ++j1)
  {
    for (int j2 = 0; j2 < 10; ++j2)
    {
      if (array[j1][j2] == 0)
      {
        goto finished;
```

```
        }

        // ...

      }
    }

  finished:             // Compliant
    return true;
  }
```

Control can also be transferred forward into a nested block by virtue of a switch label. Unless case and default labels are placed only into the top level compound statement of the switch, the code will be difficult to understand and maintain.

For Example:

```
#include <cstdint>

void f1 (int32_t i)
{
  switch (i)
  {
  case 0:
    break;
  default:
    if (i < 0)
    {
      case 1:     // Non-Compliant:  jump into a nested block
        break;
    }
    break;
  }
}
```

References:

- HIC++ v3.3 – 5.8

6.3.2 Ensure that execution of a function with a non-void return type ends in a return statement with a value

Undefined behavior will occur if execution of a function with a non void return type (other than `main`) flows off the end of the function without encountering a return statement with a value.

For Example:

```
#include <cstdint>

int32_t foo (bool b)
{
  if (b)
  {
    return -1;
  }
```

```
} // Non-Compliant:  flows off the end of the function
```

Exception:

The `main` function is exempt from this rule, as an implicit `return 0;` will be executed, when an explicit `return` statement is missing.

References:

- HIC++ v3.3 – 5.10

6.4 Declaration statement

6.4.1 Postpone variable definitions as long as possible

To preserve locality of reference, variables with automatic storage duration should be defined just before they are needed, preferably with an initializer, and in the smallest block containing all the uses of the variable.

For Example:

```
#include <cstdint>

int32_t f1 (int32_t v)
{
  int32_t i;          // Non-Compliant

  if ((v > 0) && (v < 10))
  {
    i = v * v;
    --i;
    return i;
  }
  return 0;
}

int32_t f2 (int32_t v)
{
  if ((v > 0) && (v < 10))
  {
    int32_t i (v*v); // Compliant
    --i;
    return i;
  }
  return 0;
}
```

The scope of a variable declared in a for loop initialization statement extends only to the complete for statement. Therefore, potential use of a control variable outside of the loop is naturally avoided.

For Example:

```
#include <cstdint>

int32_t f3 (int32_t max)
{
```

```
  int32_t i;
  for (i = 0; i < max; ++i)          // Non-Compliant
  {
  }
  return i;
}

void f4 (int32_t max)
{
  for (int32_t i (0); i < max; ++i) // Compliant
  {
  }
}
```

References:

- HIC++ v3.3 – 5.12
- HIC++ v3.3 – 8.4.4

7 Declarations

7.1 Specifiers

7.1.1 Declare each identifier on a separate line in a separate declaration

Declaring each variable or typedef on a separate line makes it easier to find the declaration of a particular identifier. Determining the type of a particular identifier can become confusing for multiple declarations on the same line.

For Example:

```
#include <cstdint>

// Non-Compliant:  what is the type of 'v'
extern int32_t const * p, v;

// Non-Compliant:  what type is 'Value' aliased to
typedef int32_t* Pointer, Value;
```

Exception:

For loop initialization statement is exempt from this rule, as in this context the rule conflicts with Rule 6.4.1: "Postpone variable definitions as long as possible", which takes precedence.

For Example:

```
#include <vector>
#include <cstdint>

void foo (std::vector <int32_t> const & v)
{
  for (auto iter (v.begin ()), end (v.end ()) // Compliant
    ; iter != end
    ; ++iter)
  {
    // ...
  }
}
```

References:

- HIC++ v3.3 – 8.4.2
- HIC++ v3.3 – 8.4.7

7.1.2 Use `const` whenever possible

This allows specification of semantic constraint which a compiler can enforce. It explicitly communicates to other programmers that value should remain invariant. For example, specify whether a pointer itself is const, the data it points to is const, both or neither.

For Example:

```
struct S
{
```

```
  char* p1;              // Compliant:  non-const pointer to non-const data
  const char* p2;        // Compliant:  non-const pointer to const data
  char* const p3;        // Compliant:  const pointer to non-const data
  const char* const p4; // Compliant:  const pointer to const data
};

void foo (const char * const p); // Compliant

void bar (S & s) // Non-Compliant:  parameter could be const qualified
{
  foo (s.p1);
  foo (s.p2);
  foo (s.p3);
  foo (s.p4);
}
```

Exception:

By-value return types are exempt from this rule. These should not be `const` as doing so will inhibit move semantics.

For Example:

```
struct A { };

const int f1 ();    // Non-Compliant
const A f2 ()  ;    // Non-Compliant
A f3 ();            // Compliant
```

References:

- HIC++ v3.3 – 8.4.11
- Going Native 2013 – Slide 24

7.1.3 Do not place type specifiers before non-type specifiers in a declaration

The C++ Standard allows any order of specifiers in a declaration. However, to improve readability if a non-type specifier (`typedef`, `friend`, `constexpr`, `register`, `static`, `extern`, `thread_local`, `mutable`, `inline`, `virtual`, `explicit`) appears in a declaration, it should be placed leftmost in the declaration.

For Example:

```
typedef int int32_t; // Compliant
int typedef int32_e; // Non-Compliant

class C
{
public:
  virtual inline void f1 (); // Compliant
  inline virtual void f2 (); // Compliant
  void inline virtual f3 (); // Non-Compliant

private:
  int32_t mutable _i;        // Non-Compliant
};
```

7.1.4 Place CV-qualifiers on the right hand side of the type they apply to

The const or volatile qualifiers can appear either to the right or left of the type they apply to. When the unqualified portion of the type is a typedef name (declared in a previous typedef declaration), placing the CV-qualifiers on the left hand side, may result in confusion over what part of the type the qualification applies to.

For Example:

```
#include <cstdint>

typedef int32_t * PINT;

void foo (const PINT p1 // Non-Compliant: the type is not const int32_t *
  , PINT const p2);     // Compliant: the type is int32_t * const
```

For consistency, it is recommended that this rule is applied to all declarations.

For Example:

```
#include <cstdint>

void bar (int32_t const & in); // Compliant
```

7.1.5 Do not inline large functions

The definition of an inline function needs to be available in every translation unit that uses it. This in turn requires that the definitions of inline functions and types used in the function definition must also be visible.

The `inline` keyword is just a hint, and compilers in general will only inline a function body if it can be determined that performance will be improved as a result.

As the compiler is unlikely to inline functions that have a large number of statements and expressions, inlining such functions provides no performance benefit but will result in increased dependencies between translation units.

Given an approximate cost of 1 for every expression and statement, the recommended maximum cost for a function is 32.

For Example:

```
#include <cstdint>

namespace NS
{
  class C
  {
  public:
    C (int32_t)
    {
      m_i = (m_i + m_i + m_i + m_i + m_i + m_i + m_i);
      m_i = (m_i + m_i + m_i + m_i + m_i + m_i + m_i);
      m_i = (m_i + m_i + m_i + m_i + m_i + m_i + m_i);
    }
    int32_t foo ()
    {
      m_i = (m_i + m_i + m_i + m_i + m_i + m_i + m_i);
```

```
      m_i = (m_i + m_i + m_i + m_i + m_i + m_i + m_i);
    }

  private:
    int m_i;
  };

  // Non-Compliant:  Cost greater than 32
  inline int32_t foo (int32_t i)
  {
    C c (i);
    return c.foo ();
  }
}
```

References:

- HIC++ v3.3 – 11.8

7.1.6 Use class types or typedefs to abstract scalar quantities and standard integer types

Using class types to represent scalar quantities exploits compiler enforcement of type safety. If this is not possible, typedefs should be used to aid readability of code.

For Example:

```
class Length;
class Time;
class Velocity;
class Acceleration;

// Compliant
const Velocity operator / (Length, Time);
const Velocity operator * (Acceleration, Time);
const Velocity operator * (Time, Acceleration);
```

Plain char type should not be used to define a typedef name, unless the type is intended for parameterizing the code for narrow and wide character types. In other cases, an explicit `signed char` or `unsigned char` type should be used in a typedef as appropriate.

For Example:

```
typedef char BYTE;

BYTE foo (BYTE v)
{
  return 2 * v;    // Non-Compliant:  conversion from char to integer type
}
```

To enhance portability, instead of using the standard integer types (signed char, short, int, long, long long, and the unsigned counterparts), size specific types should be defined in a project-wide header file, so that the definition can be updated to match a particular platform (16, 32 or 64bit). Where available, `intN_t` and `uintN_t` types (e.g. `int8_t`) defined in the `cstdint` header file should be used for this purpose.

For Example:

```
// Compliant:  x64 platform
typedef signed char int8_t;
typedef short       int16_t;
typedef int         int32_t;
typedef long long   int64_t;
```

Where the auto type specifier is used in a declaration, and the initializer is a constant expression, the declaration should not be allowed to resolve to a standard integer type. The type should be fixed by casting the initializer to a size specific type.

For Example:

```
#include <cstdint>

void foo ()
{
  auto a (0);      // Non-Compliant:  int
  auto b (0L);     // Non-Compliant:  long
  auto c (0U);     // Non-Compliant:  unsigned int
  auto d (static_cast <int32_t>(0)); // Compliant
  int32_t e (0);   // Compliant
}
```

Exception:

The C++ Language Standard places type requirements on certain constructs. In such cases, it is better to use required type explicitly rather than the typedef equivalent which would reduce the portability of the code.

The following constructs are therefore exceptions to this rule:

- int main()
- T operator++(int)
- T operator–(int)

For Example:

```
class A
{
public:
  A operator++(int);   // Compliant
  A operator--(int);   // Compliant
};

int main ()            // Compliant
{
}
```

References:

- HIC++ v3.3 – 8.4.5
- HIC++ v3.3 – 8.4.6

7.1.7 Use a trailing return type in preference to type disambiguation using typename

When using a trailing return type, lookup for the function return type starts from the same scope as the function declarator. In many cases, this will remove the need to specify a fully qualified return type along with the `typename` keyword.

For Example:

```
template <typename T>
class A
{
  typedef T TYPE;
  TYPE f1(TYPE);
  TYPE f2(TYPE);
};

template <typename T>
typename A<T>::TYPE A<T>::f1 (TYPE) // Non-Compliant
{
}

template <typename T>
auto  A<T>::f2 (TYPE) -> TYPE       // Compliant
{
}
```

References:

- Sutter Guru of the Week (GOTW) – 93

7.1.8 Use `auto id = expr` when declaring a variable to have the same type as its initializer function call

When declaring a variable that is initialized with a function call, the type is being specified twice. Initially on the return of the function and then in the type of the declaration.

For Example:

```
#include <cstdint>

int32_t foo();        // Type 'int32_t' specified here
int main ()
{
  int32_t i = foo(); // Non-Compliant:  Type 'int32_t' again specified here.
}
```

Using `auto` and implicitly deducing the type of the initializer will ensure that a future change to the declaration of foo will not result in the addition of unexpected implicit conversions.

For Example:

```
#include <cstdint>

int32_t foo();  // Type 'int32_t' specified here
int main ()
{
  auto i = foo(); // Compliant:  'i' deduced to have 'int32_t'.
}
```

References:

- Sutter Guru of the Week (GOTW) – 93
- 17.4.1: Use const container calls when result is immediately converted to a const iterator

7.1.9 Do not explicitly specify the return type of a lambda

Allowing the return type of a lambda to be implicitly deduced reduces the danger of unexpected implicit conversions, as well as simplifying future maintenance, where changes to types used in the lambda would otherwise result in the need to change the return type.

For Example:

```
#include <vector>
#include <algorithm>
#include <cstdint>

bool f(std::vector<int32_t> const & v1, std::vector<int32_t> & v2)
{
  v2.reserve(v1.size());
  std::transform (v1.cbegin ()
    , v1.cend ()
    , v2.begin()
    , [](std::vector<int32_t>::value_type i) -> int32_t // Non-Compliant
    { return i + 10; } );

  std::transform (v1.cbegin ()
    , v1.cend ()
    , v2.begin()
    , [](std::vector<int32_t>::value_type i)            // Compliant
    { return i + 10; } );
}
```

In the above example, if the element type of `v1` and `v2` changes, then the return type on the first lambda must also be changed, however, the second lambda will operate correctly without any update.

References:

- Sutter Guru of the Week (GOTW) – 93

7.1.10 Use `static_assert` for assertions involving compile time constants

A `static_assert` will generate a compile error if its expression is not `true`. The earlier that a problem can be diagnosed the better, with the earliest time possible being as the code is written.

For Example:

```
#include <cassert>

template <typename T, int N>
bool f(T i)
{
  // Non-Compliant
  assert((sizeof(T)*8) == N
    && "Expect␣that␣the␣size␣of␣the␣type␣matches␣the␣value␣specified␣by␣N");
```

```
  // Compliant
  static_assert((sizeof(T)*8) == N
    , "Expect that the size of the type matches the value specified by N");
}
```

7.2 Enumeration declarations

7.2.1 Use an explicit enumeration base and ensure that it is large enough to store all enumerators

The underlying type of an unscoped enumeration is implementation defined, with the only restriction being that the type must be able to represent the enumeration values. An explicit enumeration base should always be specified with a type that will accommodate both the smallest and the largest enumerator.

A scoped enum will implicitly have an underlying type of `int`, however, the requirement to specify the underlying type still applies.

For Example:

```
#include <cstdint>

enum E1                     // Non-Compliant
{
  E1_0,
  E1_1,
  E1_2
};

enum E2 : int8_t            // Compliant
{
  E2_0,
  E2_1,
  E2_2
};

enum class E3               // Non-Compliant
{
  E1_0,
  E1_1,
  E1_2
};

enum class E4 : int32_t     // Compliant
{
  E2_0,
  E2_1,
  E2_2
};
```

Exception:

An enumeration declared in an `extern "C"` block (i.e. one intended to be used with C) does not require an explicit underlying type.

For Example:

```
extern "C"
{
  enum E3         // Compliant by Exception
  {
    E3_0,
    E3_1,
    E3_2
  };
}
```

7.2.2 Initialize none, the first only or all enumerators in an enumeration

It is error prone to initialize explicitly only some enumerators in an enumeration, and to rely on the compiler to initialize the remaining ones. For example, during maintenance it may be possible to introduce implicitly initialized enumerators with the same value as an existing one initialized explicitly.

For Example:

```
#include <cstdint>
enum E : int32_t
{
  RED
  , ORANGE = 2 // Non-Compliant
  , YELLOW
};
```

Exception:

When an enumeration is used to define the size and to index an array, it is acceptable and recommended to define three additional enumerators after all other enumerators, to represent the first and the last elements, and the size of the array.

For Example:

```
#include <cstdint>

// Compliant
enum Team : int32_t
{
  Anna
  , Bob
  , Joe
  , John
  , Sandra
  , Tim
  , Team_First = Anna
  , Team_Last  = Tim
  , Team_Size
};

int32_t performance [Team_Size];
```

References:

- JSF AV C++ Rev C – 145
- MISRA C++:2008 – 8-5-3

7.3 Namespaces

7.3.1 Do not use *using directives*

Namespaces are an important tool in separating identifiers and in making interfaces explicit.

A *using directive*, i.e. `using namespace`, allows any name to be searched for in the namespace specified by the *using directive*.

A *using declaration*, on the other hand, brings in a single name from the namespace, as if it was declared in the scope containing the *using declaration*.

For Example:

```
#include <iostream>

using namespace std; // Non-Compliant
using std::cout;     // Compliant

#include <string>

// unqualified name string is looked up in namespace std
// even though the using directive precedes the
// declaration in the translation unit
string foo ();
```

As pointed out in the example above a *using directive* will affect all subsequent lookup in the translation unit. For this reason *using directives* are particularly problematic if used in header files, or occur in the main source file above a `#include` pre-processor directive. This may lead to ambiguity (compilation error) or maintenance and reuse problems.

References:

- HIC++ v3.3 – 8.2.3

7.4 Linkage specifications

7.4.1 Ensure that any objects, functions or types to be used from a single translation unit are defined in an unnamed namespace in the main source file

Declaring an entity in an unnamed namespace limits its visibility to the current translation unit only. This helps reduce the risk of name clashes and conflicts with declarations in other translation units.

It is preferred to use unnamed namespaces rather than the `static` keyword to declare such entities.

For Example:

```
static void foo (); // Non-Compliant

namespace
{
```

```
  void bar ();       // Compliant
}
```

References:

- HIC++ v3.3 – 8.3.1

7.4.2 Ensure that an inline function, a function template, or a type used from multiple translation units is defined in a single header file

An inline function, a function template or a user defined type that is intended for use in multiple translation units should be defined in a single header file, so that the definition will be processed in exactly the same way (the same sequence of tokens) in each translation unit.

This will ensure that the *one definition rule* is adhered to, avoiding undefined behavior, as well as improving the maintainability of the code.

For Example:

```
#include <cstdint>

// Non-Compliant:  definition of user defined type in the main source file
struct S
{
  int32_t i;
  int32_t j;
};
```

For Example:

```
#include <cstdint>

// Non-Compliant:  ODR violation, undefined behavior
struct S
{
  int16_t i;
  int32_t j;
};
```

References:

- HIC++ v3.3 – 8.1.2
- HIC++ v3.3 – 8.1.3

7.4.3 Ensure that an object or a function used from multiple translation units is declared in a single header file

An object or function with external linkage should be **declared** in a single header file in the project.

This will ensure that the type seen for an entity in each translation unit is the same thereby avoiding undefined behavior.

For Example:

```
// file1.cpp
// Non-Compliant:  first declaration of extern function in the main source file
```

```
extern void foo ();

// file2.cpp
// Non-Compliant:  no prior declaration in a header file
extern void foo ()
{
  // ...
}
```

References:

- HIC++ v3.3 – 8.1.1

7.5 The asm declaration

7.5.1 Do not use the `asm` declaration

Use of inline assembly should be avoided since it restricts the portability of the code.

For Example:

```
#include <cstdint>

int32_t foo ()
{
  int32_t result;

  asm ("");      // Non-Complaint

  return result;
}
```

References:

- HIC++ v3.3 – 13.5

8 Definitions

8.1 Type names

8.1.1 Do not use multiple levels of pointer indirection

In C++, at most one level of pointer indirection combined with references is sufficient to express any algorithm or API.

Instead of using multidimensional arrays, an array of containers or nested containers should be used. Code reliant on more than one level of pointer indirection will be less readable and more difficult to maintain.

For Example:

```
#include <cstdint>
#include <vector>

void foo (int32_t const * const * const pp); // Non-Compliant
void foo (int32_t const * const & rp);       // Compliant
void foo (int32_t const (& ra) [10]);        // Compliant
void foo (std::vector <int32_t> const & rv); // Compliant
```

Exception:

Use of `argv` in the `main` function is allowed.

For Example:

```
// main1.cpp
// Compliant:  argv not used
int main ();

// main2.cpp
// Compliant:  2 levels of pointer indirection in argv
int main (int argc, char * argv []);

// main3.cpp
// Compliant:  2 levels of pointer indirection in argv
int main (int argc, char * * argv);
```

References:

- JSF AV C++ Rev C – 169
- JSF AV C++ Rev C – 170
- MISRA C++:2008 – 5-0-19

8.2 Meaning of declarators

8.2.1 Make parameter names absent or identical in all declarations

Although the C++ Standard does not mandate that parameter names match in all declarations of a function (e.g. a declaration in a header file and the definition in the main source file), it is good practice to follow this principle.

For Example:

```
#include <cstdint>

void read (int32_t * buffer, int32_t * size);

void read (int32_t * size, int32_t * buffer) // Non-Compliant
{
}

class B
{
public:
  virtual void foo (int32_t in) = 0;
};

class C : public B
{
public:
  void foo (int32_t) override  // Compliant
  {
  }
};
```

References:

- HIC++ v3.3 – 11.3

8.2.2 Do not declare functions with an excessive number of parameters

A function defined with a long list of parameters often indicates poor design and is difficult to read and maintain. The recommended maximum number of function parameters is six.

For Example:

```
#include <cstdint>
#include <vector>

// Non-Compliant:  7 parameters
void foo (int32_t mode
  , int32_t const * src
  , int32_t src_size
  , int32_t * dest
  , int32_t dest_size
  , bool padding
  , bool compress);

// Compliant
void foo (int32_t flags
  , std::vector <int32_t> const & src
  , std::vector <int32_t> & dest);
```

References:

- HIC++ v3.3 – 4.3

8.2.3 Pass small objects with a trivial copy constructor by value

Because passing by const reference involves an indirection, it will be less efficient than passing by value for a small object with a trivial copy constructor.

For Example:

```
#include <cstdint>

class C
{
public:
  C (C const &) = default; // trivial copy constructor

private:
  int32_t m_i;
  int32_t m_j;
};

void foo (C v)   // Compliant
{
}

class D
{
public:
  D (D const &); // non-trivial (user defined) copy constructor

private:
  int32_t m_i;
  int32_t m_j;
};

void foo (D v)   // Non-Compliant
{
}
```

References:

- Sutter Guru of the Week (GOTW) – 91
- HIC++ v3.3 – 11.4
- HIC++ v3.3 – 11.5

8.2.4 Do not pass `std::unique_ptr` by const reference

An object of type `std::unique_ptr` should be passed as a non-const reference, or by value. Passing by non-const reference signifies that the parameter is an in/out parameter. Passing by value signifies that the parameter is a sink (i.e. takes ownership and does not return it).

A const reference `std::unique_ptr` parameter provides no benefits and restricts the potential callers of the function.

For Example:

```
#include <cstdint>
#include <memory>

void foo (std::unique_ptr<int32_t> & p_in_out);             // Compliant
void foo (std::unique_ptr<int32_t> p_sink);                 // Compliant
void foo (std::unique_ptr<int32_t> const & p_impl_detail); // Non-Compliant
```

References:

- Sutter Guru of the Week (GOTW) – 91

8.3 Function definitions

8.3.1 Do not write functions with an excessive McCabe Cyclomatic Complexity

The McCabe Cyclomatic Complexity is calculated as the number of decision branches within a function plus 1. Complex functions are hard to maintain and test effectively. It is recommended that the value of this metric does not exceed 10.

For Example:

```
#include <cstdint>

void foo (int32_t a, bool b, bool c)
{
  if (a > 0)                                    // 1
  {
    if (b)                                      // 2
    {
    }

    for (int32_t i (0); i < a; ++i)             // 3
    {
    }

    if (c)                                      // 4
    {
    }
  }
  else if (0 == a)                              // 5
  {
    if (b)                                      // 6
    {
    }

    if (c)                                      // 7
    {
    }
  }
  else
  {
    if (c)                                      // 8
```

```
    {
    }

    for (int32_t i (-a - 1); i >= 0; --i) // 9
    {
    }

    if (b)                                // 10
    {
    }
  }
  // Non-Compliant:  STCYC = #decisions + 1 = 11
}
```

References:

- HIC++ v3.3 – 4.1

8.3.2 Do not write functions with a high static program path count

Static program path count is the number of non-cyclic execution paths in a function. Functions with a high number of paths through them are difficult to test, maintain and comprehend. The static program path count of a function should not exceed 200.

For Example:

```
bool foo ();

void bar ()
{
  if (foo ()) // 2 paths
  {
  }
  if (foo ()) // 4 paths
  {
  }
  if (foo ()) // 8 paths
  {
  }
  if (foo ()) // 16 paths
  {
  }
  if (foo ()) // 32 paths
  {
  }
  if (foo ()) // 64 paths
  {
  }
  if (foo ()) // 128 paths
  {
  }
  if (foo ()) // Non-Compliant:  256 paths
  {
  }
```

```
}
```

References:

- HIC++ v3.3 – 4.2

8.3.3 Do not use default arguments

Use of default arguments can make code maintenance and refactoring more difficult. Overloaded forwarding functions can be used instead without having to change existing function calls.

For Example:

```
#include <cstdint>

void foo (int32_t i, int32_t j = 0); // Non-Compliant

// Compliant
void bar (int32_t i, int32_t j);
inline void bar (int32_t i)
{
  bar (i, 0);
}
```

References:

- 9.1.2: Make default arguments the same or absent when overriding a virtual function

8.3.4 Define `=delete` functions with parameters of type *rvalue reference* to const

A simple model for an *rvalue reference* is that it allows for the modification of a temporary. A const *rvalue reference* therefore defeats the purpose of the construct as modifications are not possible.

However, one valid use case is where the function is defined `=delete`. This will disallow the use of an *rvalue* as an argument to that function.

For Example:

```
template <class T> void f1(T&) noexcept;
template <class T> void f1(const T&&) = delete; // Compliant

template <class T> void f2(const T&&);          // Non-Compliant
```

References:

- Meyers Notes – Rvalue References and const

8.4 Initializers

8.4.1 Do not access an invalid object or an object with indeterminate value

A significant component of program correctness is that the program behavior should be deterministic. That is, given the same input and conditions the program will produce the same set of results.

If a program does not have deterministic behavior, then this may indicate that the source code is reliant on unspecified or undefined behavior.

Such behaviors may arise from use of:

- variables not yet initialized
- memory (or pointers to memory) that has been freed
- moved from objects

For Example:

```
#include <cstdint>
#include <iostream>

class A
{
public:
  A();
  // ...
};

std::ostream operator<<(std::ostream &, A const &);

int main ()
{
  int32_t i;
  A a;

  std::cout << i << std::endl; // Non-Compliant:  'i' has indeterminate value
  std::cout << a << std::endl; // Compliant:  Initialized by constructor call
}
```

Note: For the purposes of this rule, after the call to `std::move` has been evaluated the moved from argument is considered to have an indeterminate value.

For Example:

```
#include <vector>
#include <cstdint>

int main ()
{
  std::vector<int32_t> v1;
  std::vector<int32_t> v2;

  std::vector<int32_t> v3 (std::move (v1));
  std::vector<int32_t> v4 (std::move (v2));

  v1.empty (); // Non-Compliant:  'v1' considered to have indeterminate value

  v2 = v4;     // Compliant:  New value assigned to 'v2'
  v2.empty (); // before it is accessed '
}
```

References:

- 6.4.1: Postpone variable definitions as long as possible
- HIC++ v3.3 – 8.4.3
- C++11 – 8.5/11

8.4.2 Ensure that a braced aggregate initializer matches the layout of the aggregate object

If an array or a struct is non-zero initialized, initializers should be provided for all members, with an initializer list for each aggregate (sub)object enclosed in braces. This will make it clear what value each member is initialized with.

For Example:

```
#include <cstdint>

struct S
{
  int32_t i;
  int32_t j;
  int32_t k;
};

struct T
{
  struct S s;
  int32_t a[5];
};

void foo ()
{
  S s1 = {0, 1};     // Non-Compliant:  one member is not explicitly initialized
  S s2 = {0, 1, 2}; // Compliant

  T t1 = {0, 1, 2, 3, 4, 5, 6, 7};      // Non-Compliant
  T t2 = {0, 1, 2, {3, 4, 5, 6, 7}};    // Non-Compliant
  T t3 = {{0, 1, 2}, {3, 4, 5, 6, 7}}; // Compliant
}
```

References:

- JSF AV C++ Rev C – 144
- MISRA C++:2008 – 8-5-2

9 Classes

9.1 Member functions

9.1.1 Declare `static` any member function that does not require `this`. Alternatively, declare `const` any member function that does not modify the externally visible state of the object

A non-virtual member function that does not access the `this` pointer can be declared `static`. Otherwise, a function that is virtual or does not modify the externally visible state of the object can be declared `const`.

The C++ language permits that a const member function modifies the program state (e.g. modifies a global variable, or calls a function that does so). However, it is recommended that const member functions are logically const also, and do not cause any side effects.

The `mutable` keyword can be used to declare member data that can be modified in a const function, however, this should only be used where the member data does not affect the externally visible state of the object.

For Example:

```
#include <cstdint>

class C
{
public:
  explicit C (int32_t i)
    : m_i (i)
    , m_c (0)
  {
  }

  int32_t foo () // Non-Compliant:  should be static
  {
    C tmp (0);
    return tmp.bar ();
  }

  int32_t bar () // Non-Compliant:  should be const
  {
    ++ m_c;
    return m_i;
  }

private:
  int32_t m_i;
  mutable int32_t m_c;
};
```

References:

- HIC++ v3.3 – 3.1.8

9.1.2 Make default arguments the same or absent when overriding a virtual function

The C++ Language Standard allows that default arguments be different for different overrides of a virtual function. However, the compiler selects the argument value based on the static type of the object used in the function call.

This can result in confusion where the default argument value used may be different to the expectation of the user.

For Example:

```
#include <cstdint>

class Base
{
public:
  virtual void goodvFn (int32_t a = 0);
  virtual void badvFn (int32_t a = 0);
};

class Derived : public Base
{
public:
  void goodvFn (int32_t a = 0) override; // Compliant
  void badvFn (int32_t a = 10) override; // Non-Compliant
};

void foo (Derived& obj)
{
  Base& baseObj = obj;

  baseObj.goodvFn ();  // calls Derived::goodvFn with a = 0
  obj.goodvFn ();      // calls Derived::goodvFn with a = 0

  baseObj.badvFn ();   // calls Derived::badvFn with a = 0
  obj.badvFn ();       // calls Derived::badvFn with a = 10
}
```

References:

- 8.3.3: Do not use default arguments
- HIC++ v3.3 – 3.3.12

9.1.3 Do not return non-const handles to class data from const member functions

A pointer or reference to non-const data returned from a const member function may allow the caller to modify the state of the object. This contradicts the intent of a const member function.

For Example:

```
#include <cstdint>

class C
{
public:
  C () : m_i (new int32_t) {}

  ~C()
  {
```

```
    delete m_i;
  }

  int32_t * get () const
  {
    return m_i;  // Non-Compliant
  }

private:
  int32_t * m_i;

  C (C const &) = delete;
  C & operator = (C const &) & = delete;
};
```

Exception:

Resource handler classes that do not maintain ownership of a resource are exempt from this rule, as in this context the rule conflicts with Rule 9.1.1: "Declare `static` any member function that does not require `this`. Alternatively, declare `const` any member function that does not modify the externally visible state of the object", which takes precedence.

For Example:

```
#include <cstdint>

class D
{
public:
  D (int32_t * p) : m_i (p) {}

  int32_t * get () const
  {
    return m_i;  // Compliant
  }

private:
  int32_t * m_i;
};
```

References:

- HIC++ v3.3 – 3.4.2

9.1.4 Do not write member functions which return non-const handles to data less accessible than the member function

Member data that is returned by a non-const handle from a more accessible member function, implicitly has the access of the function and not the access it was declared with. This reduces encapsulation and increases coupling.

For Example:

```
#include <cstdint>

class C
```

```
{
public:
  C () : m_i (0) {}

  int32_t & get ()          // Non-Compliant
  {
    return m_i;
  }

  int const & get () const // Compliant
  {
    return m_i;
  }

private:
  int32_t m_i;
};
```

Exception:

Non-const `operator []` is exempt from this rule, as in this context the rule conflicts with Rule 13.2.4: "When overloading the subscript operator (`operator[]`) implement both const and non-const versions", which takes precedence.

For Example:

```
#include <cstdint>

class Array
{
public:
  Array () ;

int32_t & operator [] (int32_t a)      // Compliant:  non-const version
{
  return m_x[ a ];
}

int32_t operator [] (int32_t a) const // Compliant:  const version
{
  return m_x[ a ];
}

private:
  static const int32_t Max_Size = 10;
  int32_t m_x [Max_Size];
};
```

References:

- HIC++ v3.3 – 3.4.3

9.1.5 Do not introduce virtual functions in a final class

Declaring a class as `final` explicitly documents that this is a leaf class as it cannot be used as a base class. Introducing a virtual function in such a class is therefore redundant as the function can never be overridden in a derived class.

For Example:

```
class Base
{
public:
  virtual void f1 ();    // Compliant
};

class Derived final : public Base
{
public:
  virtual void f2 ();    // Non-Compliant
};
```

9.2 Bit-fields

9.2.1 Declare bit-fields with an explicitly unsigned integral or enumeration type

To avoid reliance on implementation defined behavior, only declare bit-fields of an explicitly unsigned type (`uintN_t`) or an enumeration type with an enumeration base of explicitly unsigned type.

For Example:

```
#include <cstdint>
enum E : uint8_t { ONE, TWO, THREE };
struct S
{
  int32_t a : 2; // Non-Compliant
  uint8_t b : 2; // Compliant
  bool    c : 1; // Non-Compliant
  char    d : 2; // Non-Compliant
  wchar_t e : 2; // Non-Compliant
  E       f : 2; // Compliant
};
```

References:

- JSF AV C++ Rev C – 154
- MISRA C++:2008 – 9-6-2
- MISRA C++:2008 – 9-6-3

10 Derived classes

10.1 Multiple base classes

10.1.1 Ensure that access to base class subobjects does not require explicit disambiguation

A class inherited more than once in a hierarchy, and not inherited virtually in all paths will result in multiple base class subobjects being present in instances of the derived object type.

Such objects require that the developer explicitly select which base class to use when accessing members. The result is a hierarchy that is harder to understand and maintain.

For Example:

```
class Base
{
public:
  void foo ();
};

class Derived_left: public Base {};

class Derived_right: public Base {};

// Non-Compliant:  2 subobjects for 'Base'
class Derived: public Derived_left, public Derived_right
{
};

void test()
{
  Derived d;
  // ambiguous - Derived_left::Base::foo or Derived_right::Base::foo?
  d.foo ();
}
```

The example above can be made to comply with this rule by using virtual inheritance:

For Example:

```
class Base {};

class Derived_left: public virtual Base {};

class Derived_right: public virtual Base {};

class Derived: public Derived_left, public Derived_right // Compliant
{
};
```

References:

- HIC++ v3.3 – 3.3.15

10.2 Virtual functions

10.2.1 Use the `override` special identifier when overriding a virtual function

The `override` special identifier is a directive to the compiler to check that the function is overriding a base class member. This will ensure that a change in the signature of the virtual function will generate a compiler error.

For Example:

```
#include <cstdint>

class A {
public:
  virtual void f(int64_t);
};

class B : public A {
public:
  void f(int32_t) override;    // Compliant:  Compile Error
};
```

Note: The following was considered good C++ '03 style:

For Example:

```
#include <cstdint>

class A {
public:
  virtual void f(int32_t);
};

class B : public A {
public:
  virtual void f(int32_t);    // Non-Compliant
};

void f(A & a)
{
  a.f(0);                  // Results in B::f being called
}
```

However, this provided no guarantees and no additional checking by the compiler if the function signature was changed.

For Example:

```
#include <cstdint>

class A {
public:
  virtual void f(int64_t);
};

class B : public A {
```

```
public:
  virtual void f(int32_t);    // Non-Compliant
};

void f(A & a)
{
  a.f(0);                  // Results in A::f being called!!!!
}
```

References:

- HIC++ v3.3 – 3.3.16

10.3 Abstract classes

10.3.1 Ensure that a derived class has at most one base class which is not an interface class

An interface class has the following properties:

- 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

Inheriting from two or more base classes that are not interfaces, is rarely correct. It also exposes the derived class to multiple implementations, with the risk that subsequent changes to any of the base classes may invalidate the derived class.

On the other hand. it is reasonable that a concrete class may implement more than one interface.

For Example:

```
#include <cstdint>

class A
{
public:
  virtual ~A () = 0;
  virtual void foo () = 0;
};

class B
{
public:
  virtual ~B () = 0;
  virtual void bar () = 0;
};

class C
{
public:
  C ();
  void foo ();
  virtual ~C ();
```

```
private:
  int32_t m_i;
};

// Compliant
class D: public A, public B, public C
{
public:
  ~ D();
};

class E
{
public:
  E ();
};

// Non-Compliant
class F : public E, public D
{
};
```

References:

- JSF AV C++ Rev C – 87
- JSF AV C++ Rev C – 88
- HIC++ v3.3 – 3.4.6

11 Member access control

11.1 Access specifiers

11.1.1 Declare all data members private

If direct access to the object state is allowed through public or protected member data, encapsulation is reduced making the code harder to maintain.

By implementing a class interface with member functions only, precise control is achieved over modifications to object state as well as allowing for pre and post conditions to be checked when accessing data.

For Example:

```
#include <string>
#include <cassert>

class C
{
public:
  C ();

  std::string m_id; // Non-Compliant
};

class D
{
public:
  D ();

  std::string const & getId () const
  {
    assert (! m_id.empty () && "Id not yet specified");
    return m_id;
  }

private:
  std::string m_id; // Compliant
};
```

Exception:

Class types declared with the class key `struct` in an `extern "C"` block are intended to be compatible with C, and therefore such definitions are not covered by this rule.

For Example:

```
extern "C"
{
  struct S    // Compliant
  {
    int i;
    int j;
  };
```

```
}
```

References:

- HIC++ v3.3 – 3.4.1

11.2 Friends

11.2.1 Do not use friend declarations

Friend declarations reduce encapsulation, resulting in code that is harder to maintain.

For Example:

```
class C
{
public:
  C & operator += (C const & other);

  friend C const operator + (C const &, C const & lhs); // Non-Compliant
};

class D
{
public:
  D & operator += (D const & other);
};

D const operator + (D const & rhs, D const & lhs) // Compliant
{
  D result (rhs);
  result += (lhs);
  return result;
}
```

The following example demonstrates an additional complexity when adding friendship for functions that include universal reference parameters. In such cases an explicit declaration is required for all combinations of the *lvalue reference* and *rvalue reference* versions.

For Example:

```
template <typename T, typename S>
void foo (T && t, S && s)
{
  t.foo ();
}

class A
{
private:
  void foo ();

  friend void foo(A &, A &);
  friend void foo(A &, A&&);
```

```
  friend void foo(A&&, A &);
  friend void foo(A&&, A&&);
};

int main ()
{
  A a;
  foo(a, a);
  foo(a, A());
  foo(A(), a);
  foo(A(), A());
}
```

Exception:

In some cases friend declarations are part of the class interface, for example:

- serialization via stream input and output operators (`operator <<` and `operator >>`)
- factory functions requiring access to private constructors
- an iterator class

For Example:

```
#include <cstdint>

class C
{
public:
  C (C const &) = default;
  C (C &&) = default;

private:
  C (int32_t);
  friend C createClassC(int32_t);
};

C createClassC (int32_t i)
{
  // pre condition check on 'i'
  return C(i);
}
```

References:

- HIC++ v3.3 – 3.4.4

12 Special member functions

12.1 Conversions

12.1.1 Do not declare implicit user defined conversions

A user defined conversions can occur through the use of a conversion operator or a conversion constructor (a constructor that accepts a single argument).

A compiler can invoke a single user defined conversion in a standard conversion sequence, but only if the operator or constructor is declared without the `explicit` keyword.

It is better to declare all conversion constructors and operators explicit.

For Example:

```
#include <cstdint>

class C
{
public:
  C (const C&);                          // Compliant:  copy constructor
  C ();                                  // Compliant:  default constructor
  C (int32_t, int32_t);                  // Compliant:  more than one non-default argument


  explicit C (int32_t);                  // Compliant
  C (double);                            // Non-Compliant
  C (float f, int32_t i = 0);            // Non-Compliant
  C (int32_t i = 0, float f = 0.0);      // Non-Compliant:  default constructor,
                                         // but also a conversion constructor
  operator int32_t () const;             // Non-Compliant
  explicit operator double () const; // Compliant
};
```

References:

- HIC++ v3.3 – 3.2.3
- HIC++ v3.3 – 3.1.10
- HIC++ v3.3 – 3.1.11

12.2 Destructors

12.2.1 Declare `virtual`, `private` or `protected` the destructor of a type used as a base class

If an object will ever be destroyed through a pointer to its base class, then the destructor in the base class should be `virtual`. If the base class destructor is not virtual, then the destructors for derived classes will not be invoked.

Where an object will not be deleted via a pointer to its base, then the destructor should be declared with `protected` or `private` access. This will result in a compile error should an attempt be made to delete the object incorrectly.

For Example:

```
#include <cstdint>

class A
{
public:
  ~A ();          // Non-Compliant
};

class B : public A
{
};

class C
{
public:
  virtual ~C (); // Compliant
};

class D : public C
{
};

class E
{
protected:
  ~E ();          // Compliant
};

class F : public E
{
};
```

References:

- HIC++ v3.3 – 3.3.2

12.3 Free store

12.3.1 Correctly declare overloads for `operator new` and `delete`

`operator new` and `operator delete` should work together. Overloading `operator new` means that a custom memory management scheme is in operation for a particular class or program. If a corresponding `operator delete` (plain or array) is not provided the memory management scheme is incomplete.

Additionally, if initialization of the allocated object fails with an exception, the C++ runtime will try to call an `operator delete` with identical parameters as the called `operator new`, except for the first parameter. If no such `operator delete` can be found, the memory will not be freed. If this `operator delete` does not actually need to perform any bookkeeping, one with an empty body should be defined to document this in the code.

When declared in a class, `operator new` and `operator delete` are implicitly static members; explicitly including the `static` specifier in their declarations helps to document this.

For Example:

```
#include <cstddef>

class C
{
public:
  static void* operator new (std::size_t size);
  static void operator delete (void* ptr);    // Compliant

  void* operator new [] (std::size_t size);  // Non-Compliant
};
```

References:

- HIC++ v3.3 – 12.6
- HIC++ v3.3 – 12.7

12.4 Initializing bases and members

12.4.1 Do not use the dynamic type of an object unless the object is fully constructed

Expressions involving:

- a call to a virtual member function,
- use of `typeid`, or
- a cast to a derived type using `dynamic_cast`

are said to use the dynamic type of the object.

Special semantics apply when using the dynamic type of an object while it is being constructed or destructed. Moreover, it is undefined behavior if the static type of the operand is not (or is not a pointer to) the constructor's or destructor's class or one of its base classes.

In order to avoid misconceptions and potential undefined behavior, such expressions should not be used while the object is being constructed or destructed.

For Example:

```
#include <typeinfo>
class A
{
public:
  virtual void foo ();
  virtual void bar ();
};

class B : public A
{
public:
  B();
  ~B();

  void foo () override;
```

```
};

class C : public B
{
public:
  void foo () override;
  void bar () override;
};

B::B()
{
  foo ();                       // Non-Compliant:  B::foo () is called and never C::foo ()
  B::foo ();                    // Compliant:  not a virtual call
  typeid (*this);               // Non-Compliant
}

B::~B()
{
  bar ();                       // Non-Compliant:  A::bar () is called and never C::bar ()
  A::bar ();                    // Compliant:  not a virtual call
  dynamic_cast <A*> (this); // Compliant:  dynamic type is not needed for a downcast
  dynamic_cast <C*> (this); // Non-Compliant
}

void foo ()
{
  C c;
}
```

References:

- HIC++ v3.3 – 3.3.13
- JSF AV C++ Rev C – 71
- MISRA C++:2008 – 12-1-1

12.4.2 Ensure that a constructor initializes explicitly all base classes and non-static data members

A constructor should completely initialize its object. Explicit initialization reduces the risk of an invalid state after successful construction. All virtual base classes and direct non-virtual base classes should be included in the initialization list for the constructor. A copy or move constructor should initialize each non-static data member in the initialization list, or if this is not possible then in constructor body. For other constructors, each non-static data member should be initialized in the following way, in order of preference:

- non static data member initializer (NSDMI), or
- in initialization list, or
- in constructor body.

For many constructors this means that the body becomes an empty block.

For Example:

```
class C
{
```

```
public:
  C () : m_j (0), m_a () {}            // Compliant

  // Non-Compliant:  m_a not initialized
  C (C const & other) : m_i (other.m_i), m_j (other.m_j) {}

  explicit C (int32_t j) : m_j (j) {} // Non-Compliant:  m_a not initialized

private:
  int32_t m_i = 0;
  int32_t m_j;
  int32_t m_a [10];
};
```

References:

- HIC++ v3.3 – 3.2.1

12.4.3 Do not specify both an NSDMI and a member initializer in a constructor for the same non static member

NSDMI stands for 'non static data member initializer'. This syntax, introduced in the 2011 C++ Language Standard, allows for the initializer of a member to be specified along with the declaration of the member in the class body. To avoid confusion as to the value of the initializer actually used, if a member has an NSDMI then it should not subsequently be initialized in the member initialization list of a constructor.

For Example:

```
#include <cstdint>

class A
{
public:
  A()
  : m_i1(1)          // Compliant
  , m_i2(1)          // Non-Compliant
  {
  }

private:
  int m_i1;
  int m_i2 = 0;      // Non-Compliant
  int m_i3 = 0;      // Compliant
};
```

Exception:

The move/copy constructors are exempt from this rule, as in this context the rule conflicts with Rule 12.4.2: "Ensure that a constructor initializes explicitly all base classes and non-static data members", which takes precedence.

For Example:

```
#include <cstdint>

class A
```

```
{
public:
  A(A const & rhs)
  : m_i1(rhs.m_i1)          // Compliant
  , m_i2(rhs.m_i2)          // Compliant
  {
  }

private:
  int32_t m_i1;
  int32_t m_i2 = 0;
};
```

12.4.4 Write members in an initialization list in the order in which they are declared

Regardless of the order of member initializers in a initialization list, the order of initialization is always:

- Virtual base classes in depth and left to right order of the inheritance graph.
- Direct non-virtual base classes in left to right order of inheritance list.
- Non-static member data in order of declaration in the class definition.

To avoid confusion and possible use of uninitialized data members, it is recommended that the initialization list matches the actual initialization order.

For Example:

```
#include <cstdint>

class B {};

class VB : public virtual B {};

class C {};

class DC : public VB, public C
{
public:
  DC()
    : B(), VB(), C(), i (1), c() // Compliant
  {}

private:
  int32_t i;
  C c;
};
```

References:

- HIC++ v3.3 – 3.2.2

12.4.5 Use delegating constructors to reduce code duplication

Delegating constructors can help reduce code duplication by performing initialization in a single constructor. Using delegating constructors also removes a potential performance penalty with using an 'init' method, where initialization for some members occurs twice.

For Example:

```
#include <cstdint>

// Non-Compliant
class A1 {
public:
  A1()
  {
    init(10, 20);
  }

  A1(int i)
  {
    init(i, 20);
  }
private:
  void init(int32_t i, int32_t j);

private:
  int32_t m_i;
  int32_t m_j;
};

// Compliant
class A2 {
public:
  A2()
  : A2(10, 20)
  {
  }

  A2(int32_t i)
  : A2(i, 20)
  {
  }

private:
  A2(int32_t i, int32_t j)
  : m_i(i)
  , m_j(j)
  {
  }

private:
  int32_t m_i;
  int32_t m_j;
};
```

12.5 Copying and moving class objects

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

A compiler may provide some or all of the following special member functions:

- Destructor
- Copy constructor
- Copy assignment operator
- Move constructor
- Move assignment operator

The set of functions implicitly provided depends on the special member functions that have been declared by the user and also the special members of base classes and member objects.

The compiler generated versions of these functions perform a bitwise or shallow copy, which may not be the correct copy semantics for the class. It is also not clear to clients of the class if these functions can be used or not.

To resolve this, the functions should be defined with =delete or =default thereby fully documenting the class interface.

For Example:

```
#include <cstdint>

class A1      // Non-Compliant:  Includes implicitly declared special members
{
public:
  A1();
  ~A1();

private:
  int32_t * m_i;
};

class A2      // Compliant:  No implicitly declared special members
{
public:
  A2();
  ~A2();
  A2(A2 const &) = default;
  A2 & operator=(A2 const &) & = delete;

private:
  int32_t * m_i;
};
```

Note: As this rule is limited to concrete classes, it is the responsibility of the most derived class to ensure that the object has correct copy semantics for itself and for its sub-objects.

References:

- HIC++ v3.3 – 3.1.3
- HIC++ v3.3 – 3.1.13

12.5.2 Define special members `=default` if the behavior is equivalent

The implicitly defined copy constructor for a class X with two bases and two members will be defined as:

Copy Constructor:

```
X::X(X const & rhs)
: base1(rhs)
, base2(rhs)
, mbr1(rhs.mbr1)
, mbr2(rhs.mbr2)
{
}
```

The implicitly defined move constructor for the same class X will be defined as:

Move Constructor:

```
X::X(X && rhs)
: base1(std::move(rhs))
, base2(std::move(rhs))
, mbr1(std::move(rhs.mbr1))
, mbr2(std::move(rhs.mbr2))
{
}
```

Finally, the implicitly defined destructor will be defined as:

Destructor:

```
X::~X()
{
}
```

If a class contains a user defined version of a member with the same definition as would be provided by the compiler, then it will be less error prone and more maintainable to replace the definition with `=default`.

For Example:

```
#include <cstdint>

class A
{
public:
  ~A()                  // Non-Compliant
  {
  }

  A(A const & rhs)      // Non-Compliant
  : mbr(rhs.mbr)
  {
```

```
  }

  A(A &&) = default;  // Compliant

private:
  int32_t mbr;
};
```

References:

- JSF AV C++ Rev C – 80

12.5.3 Ensure that a user defined move/copy constructor only moves/copies base and member objects

The human clients of a class will expect that the copy constructor can be used to correctly copy an object of class type. Similarly, they will expect that the move constructor correctly moves an object of class type.

Similarly, a compiler has explicit permission in the C++ Standard to remove unnecessary copies or moves, on the basis that these functions have no other side-effects other than to copy or move all bases and members.

For Example:

```
#include <cstdint>
#include <utility>

class Base
{
public:
  Base ()
    : m_j (-1)
  {
  }

  Base (Base const & rhs)
    : m_j (rhs.m_j)
  {
  }

  Base (Base && rhs) noexcept
    : m_j (std::move (rhs.m_j))
  {
  }

private:
  int32_t m_j;
};

void foo ();

class Derived1 : public Base
{
public:
  Derived1 (Derived1 const & rhs)
    : Base (rhs)
```

```
    , m_i (rhs.m_i)
  {
    foo (); // Non-Compliant:  unrelated side effect
  }

  Derived1 (Derived1 && rhs) noexcept
    : Base (std::move (rhs))
    , m_i (std::move (rhs.m_i))
  {
    foo (); // Non-Compliant:  unrelated side effect
  }

private:
  int32_t m_i;
};

class Derived2 : public Base
{
public:
  Derived2 (Derived2 const & rhs)       // Compliant
    : Base (rhs)
    , m_i (rhs.m_i)
  {
  }

  Derived2 (Derived2 && rhs) noexcept   // Compliant
    : Base (std::move (rhs))
    , m_i (std::move (rhs.m_i))
  {
  }

private:
  int32_t m_i;
};
```

References:

- MISRA C++:2008 – 12-8-1

12.5.4 Declare `noexcept` the move constructor and move assignment operator

A class provides the Strong Exception Guarantee if after an exception occurs, the objects maintain their original values.

The move members of a class explicitly change the state of their argument. Should an exception be thrown after some members have been moved, then the Strong Exception Guarantee may no longer hold as the from object has been modified.

It is especially important to use `noexcept` for types that are intended to be used with the standard library containers. If the move constructor for an element type in a container is not `noexcept` then the container will use the copy constructor rather than the move constructor.

For Example:

```
#include <utility>
```

```
#include <cstdint>

class B
{
public:
  B (B && rhs) noexcept          // Compliant
  : m_p (std::move (rhs.m_p))
  {
    rhs.m_p = 0;
  }

private:
  int32_t * m_p;
};

class A
{
public:
  A (A && rhs)                   // Non-Compliant
  : m_a (std::move (rhs.m_a))
  , m_b ((rhs.m_b) ? std::move (rhs.m_b) : throw 0)
  {
  }

  A& operator = (A&&) noexcept; // Compliant

private:
  B m_a;
  int32_t m_b;
};
```

12.5.5 Correctly reset moved-from handles to resources in the move constructor

The move constructor moves the ownership of data from one object to another. Once a resource has been moved to a new object, it is important that the moved-from object has its handles set to a default value. This will ensure that the moved-from object will not attempt to destroy resources that it no longer manages on its destruction.

The most common example of this is to assign `nullptr` to pointer members.

For Example:

```
#include <utility>
#include <cstdint>

class A1
{
public:
  A1(A1 && rhs) noexcept          // Non-Compliant
  : m_p(std::move(rhs.m_p))
  {
  }

  ~A1()
```

```
  {
    delete m_p;                    // Moved-from object will attempt to delete
                                   // resource owned by other object
  }

private:
  int32_t * m_p;
};

class A2
{
public:
  A2(A2 && rhs) noexcept           // Compliant
  : m_p(std::move(rhs.m_p))
  {
    rhs.m_p = nullptr;
  }

  ~A2()
  {
    delete m_p;                    // Moved-from object will not delete
                                   // resource owned by other object
  }

private:
  int32_t * m_p;
};
```

12.5.6 Use an atomic, non-throwing swap operation to implement the copy and move assignment operators

Implementing the copy assignment operator using a non throwing swap provides the Strong Exception Guarantee for the operations.

In addition, the implementation of each assignment operator is simplified without requiring a check for assignment to self.

For Example:

```
#include <utility>
#include <cstdint>

class A1
{
public:
  A1(A1 const & rhs)
  : m_p1(new int32_t (*rhs.m_p1))
  , m_p2(new int32_t (*rhs.m_p2))
  {
  }

  A1(A1 && rhs) noexcept
  : m_p1(std::move (rhs.m_p1))
```

```
  , m_p2(std::move (rhs.m_p2))
  {
    rhs.m_p1 = nullptr;
    rhs.m_p2 = nullptr;
  }

  ~A1()
  {
    delete m_p1;
    delete m_p2;
  }

  A1 & operator=(A1 const & rhs) &      // Non-Compliant
  {
    if (this != &rhs)
    {
      m_p1 = new int32_t (*rhs.m_p1);

      // An exception here would result in a memory leak for m_p1
      m_p2 = new int32_t (*rhs.m_p2);
    }
    return *this;
  }

  A1 & operator=(A1 && rhs) & noexcept // Non-Compliant
  {
    if (this != &rhs)
    {
      m_p1 = std::move (rhs.m_p1);
      m_p2 = std::move (rhs.m_p2);
      rhs.m_p1 = nullptr;
      rhs.m_p2 = nullptr;
    }
    return *this;
  }

private:
  int32_t * m_p1;
  int32_t * m_p2;
};

class A2
{
public:
  A2(A2 const & rhs)
  : m_p1(new int32_t (*rhs.m_p1))
  , m_p2(new int32_t (*rhs.m_p2))
  {
  }

  A2(A2 && rhs) noexcept
  : m_p1(std::move (rhs.m_p1))
```

```
  , m_p2(std::move (rhs.m_p2))
  {
    rhs.m_p1 = nullptr;
    rhs.m_p2 = nullptr;
  }

  ~A2()
  {
    delete m_p1;
    delete m_p2;
  }

  A2 & operator=(A2 rhs) &              // Compliant: Note: 'rhs' is by value
  {
    swap (*this, rhs);
    return *this;
  }

  A2 & operator=(A2 && rhs) & noexcept // Compliant
  {
    A2 tmp (std::move (rhs));
    swap (*this, tmp);
    return *this;
  }

  void swap(A2 & lhs, A2 & rhs) noexcept
  {
    std::swap (lhs.m_p1, rhs.m_p1);
    std::swap (lhs.m_p2, rhs.m_p2);
  }

private:
  int32_t * m_p1;
  int32_t * m_p2;
};
```

References:

- HIC++ v3.3 – 3.1.4

12.5.7 Declare assignment operators with the ref-qualifier &

In the 2003 C++ Language Standard, user declared types differed from built-in types in that it was possible to have a 'modifiable rvalue'.

For Example:

```
#include <cstdint>

class A {
public:
  A();
  A & operator*=(int32_t);     // Non-Compliant
};
```

```
A f1();
int32_t f2();
int main ()
{
  f1() *= 10;     // Temporary result of 'f()' multiplied by '10'
  f2() *= 10;     // Compile error
}
```

The 2011 C++ Language Standard allows for a function to be declared with a reference qualifier. Adding `&` to the function declaration ensures that the call can only be made on *lvalue* objects, as is the case for the built-in operators.

For Example:

```
#include <cstdint>

class A {
public:
  A();
  A & operator*=(int32_t) &;   // Compliant
};
A f1();
int32_t f2();
int main ()
{
  f1() *= 10;     // Compile error
  f2() *= 10;     // Compile error
}
```

References:

- HIC++ v3.3 – 3.1.5

12.5.8 Make the copy assignment operator of an abstract class protected or define it `=delete`

An instance of an abstract class can only exist as a subobject for a derived type. A public copy assignment operator would allow for incorrect partial assignments to occur.

The copy assignment operator should be protected, or alternatively defined `=delete` if copying is to be prohibited in this class hierarchy.

For Example:

```
#include <cstdint>

class A
{
public:
  virtual ~A () = 0;
  A& operator = (A const &) &;            // Non-Compliant
};

class AA : public A
{
};
```

```
void foo()
{
  AA obj1;
  AA obj2;

  A* ptr1 = &obj1;
  A* ptr2 = &obj2;

  *ptr1 = *ptr2; // partial assignment
}

class B
{
public:
  virtual ~B () = 0;

protected:
  B& operator = (B const &) &;            // Compliant
};

class C
{
public:
  virtual ~C () = 0;

protected:
  C& operator = (C const &) & = default; // Compliant
};

class D
{
public:
  virtual ~D () = 0;
  D& operator = (D const &) & = delete;  // Compliant
};
```

References:

- HIC++ v3.3 – 3.3.14

13 Overloading

13.1 Overload resolution

13.1.1 Ensure that all overloads of a function are visible from where it is called

When a member function is overridden or overloaded in a derived class, other base class functions of that name will be hidden. A call to a function from the derived class may therefore result in a different function being called than if the same call had taken place from the base class.

To avoid this situation, hidden names should be introduced into the derived class through a *using declaration*.

For Example:

```
#include <cstdint>

class B
{
public:
  void foo (uint32_t);

  virtual void bar (uint32_t);
  virtual void bar (double);
};

class D : public B
{
public:
  void foo (double);           // Non-Compliant

  void bar (double) override; // Non-Compliant
};

void f1 ()
{
  D d;
  d.foo (0U); // D::foo (double) called
  B & b (d);
  b.foo (0U); // B::foo (uint32_t) called

  d.bar (0U); // D::bar (double) called
  b.bar (0U); // B::bar (uint32_t) called
}

class E : public B
{
public:
  using B::foo;
  void foo (double);           // Compliant

  using B::bar;
  void bar (double) override; // Compliant
```

```
};

void f2 ()
{
  E d;
  d.foo (0U); // B::foo (uint32_t) called
  B & b (d);
  b.foo (0U); // B::foo (uint32_t) called

  d.bar (0U); // B::bar (uint32_t) called
  b.bar (0U); // B::bar (uint32_t) called
}
```

A using declaration for a namespace scope identifier, only brings into the current scope the prior declarations of this identifier, and not any declarations subsequently added to the namespace. This too may lead to unexpected results for calls to overloaded functions.

For Example:

```
#include <cstdint>

namespace NS
{
  void foo (int32_t);
  struct A
  {
    int32_t a;
    int32_t b;
  };
}

using NS::foo;
using NS::A;

namespace NS
{
  void foo (uint32_t);
  int A;
}

uint32_t bar (uint32_t u)
{
  foo (u);            // Non-Compliant:  foo (int32_t) called
  return sizeof (A); // Non-Compliant:  evaluates sizeof (struct A)
}
```

References:

- HIC++ v3.3 – 3.3.5
- HIC++ v3.3 – 3.3.11
- MISRA C++:2008 – 7-3-5

13.1.2 If a member of a set of callable functions includes a universal reference parameter, ensure that one appears in the same position for all other members

A callable function is one which can be called with the supplied arguments. In the C++ Language Standard, this is known as the set of viable functions.

A template parameter declared `T&&` has special rules during type deduction depending on the value category of the argument to the function call. Scott Meyers has named this a 'Universal Reference'.

For Example:

```
#include <cstdint>

template<typename T>
void f1 (T&&t);
void f2 ()
{
  int32_t i;
  f1(i); // 't' has type int &, T has type 'int &'
  f1(0); // 't' has type int &&, T has type 'int'
}
```

As a universal reference will deduce perfectly for any type, overloading them can easily lead to confusion as to which function has been selected.

For Example:

```
#include <cstdint>

template <typename T> void f1 (T&&t);   // #1 // Not Compliant
void f1 (int&&t);                       // #2

void f2()
{
  int32_t i = 0;
  f1(i);                                // Calls #1
  f1(+i);                               // Calls #2
  f1(0);                                // Calls #2
  f1(0U);                               // Calls #1
}
```

Exception:

Standard C++ allows for a member of the viable function set to be deleted. In such cases, should these functions be called then it will result in a compiler error.

For Example:

```
#include <cstdint>

template<typename T>
void f (T&&t);

void f (int32_t &) = delete;     // Compliant
```

```
int main ()
{
  int32_t i;
  f (0);
  f (i);
}
```

References:

- Meyers Effective C++ '11 (draft TOC) – Avoid overloading on universal references

13.2 Overloaded operators

13.2.1 Do not overload operators with special semantics

Overloaded operators are just functions, so the order of evaluation of their arguments is unspecified. This is contrary to the special semantics of the following built-in operators:

- `&&` – left to right and potentially evaluated
- `||` – left to right and potentially evaluated
- `,` – left to right

Providing user declared versions of these operators may lead to code that has unexpected behavior and is therefore harder to maintain.

For Example:

```
class A
{
public:
  bool operator && (A const &);           // Non-Compliant
};

bool operator || (A const &, A const &); // Non-Compliant
A operator , (A const &, A const &);     // Non-Compliant
```

Additionally, overloading the unary `&` (address of) operator will result in undefined behavior if the operator is used from a location in the source where the user provided overload is not visible.

For Example:

```
class A;

A * foo (A & a)
{
  return & a;         // a.operator& not visible here
}

class A
{
public:
  A * operator & (); // Non-Compliant:  undefined behavior
};
```

References:

- HIC++ v3.3 – 3.5.1
- JSF AV C++ Rev C – 159
- JSF AV C++ Rev C – 168
- MISRA C++:2008 – 5-3-3
- MISRA C++:2008 – 5-2-11
- MISRA C++:2008 – 5-18-1

13.2.2 Ensure that the return type of an overloaded binary operator matches the built-in counterparts

Built-in binary arithmetic and bitwise operators return a pure *rvalue* (which cannot be modified), this should be mirrored by the overloaded versions of these operators. For this reason the only acceptable return type is a fundamental or an enumerated type or a class type with a reference qualified assignment operator.

Built-in equality and relational operators return a boolean value, and so should the overloaded counterparts.

For Example:

```
#include <cstdint>

class A
{
public:
  A & operator=(A const &) &;
  // ...
};

A operator + (A const &, A const &);        // Compliant
const A operator - (A const &, A const &); // Non-Compliant
A & operator | (A const &, A const &);      // Non-Compliant
bool operator == (A const &, A const &);    // Compliant
int32_t operator < (A const &, A const &); // Non-Compliant
```

References:

- HIC++ v3.3 – 3.5.3

13.2.3 Declare binary arithmetic and bitwise operators as non-members

Overloaded binary arithmetic and bitwise operators should be non-members to allow for operands of different types, e.g. a fundamental type and a class type, or two unrelated class types.

For Example:

```
#include <iostream>

class A
{
public:
  bool operator * (A const & other);                    // Non-Compliant
  bool operator == (A const & other);                   // Compliant
};
```

```
A operator + (int32_t lhs, A const & rhs);                    // Compliant
A operator + (A const & lhs, int32_t rhs);                    // Compliant
std::ostream & operator << (std::ostream & o, A const & a); // Compliant
```

References:

- HIC++ v3.3 – 3.5.4

13.2.4 When overloading the subscript operator (operator[]) implement both const and non-const versions

A non-const overload of the subscript operator should allow an object to be modified, i.e. should return a reference to member data. The const version is there to allow the operator to be invoked on a const object.

For Example:

```
#include <cstdint>

class Array
{
public:
  Array ()
  {
    for (int32_t i = 0; i < Max_Size; ++i )
    {
      m_x [i] = i;
    }
  }

  int32_t & operator [] (int32_t a)      // Compliant:  non-const version
  {
    return m_x[ a ];
  }

  int32_t operator [] (int32_t a) const // Compliant:  const version
  {
    return m_x[ a ];
  }

private:
  static const int32_t Max_Size = 10;
  int32_t m_x [Max_Size];
};

void foo ()
{
  Array a;
  int32_t i = a [3];    // non-const
  a [3] = 33;           // non-const

  Array const ca;
  i = ca [3];           // const
  ca [3] = 33;          // compilation error
```

```
}
```

References:

- HIC++ v3.3 – 3.5.5

13.2.5 Implement a minimal set of operators and use them to implement all other related operators

In order to limit duplication of code and associated maintenance overheads, certain operators can be implemented in terms of other operators.

Binary Arithmetic and Bitwise Operators

Each binary arithmetic or bitwise operator can be implemented in terms of its compound assignment counterpart.

For Example:

```
class A
{
public:
  A & operator += (A const & other);
};

A const operator + (A const & lhs, A const & rhs) // Compliant
{
  A result (lhs);
  result += rhs;
  return result;
}
```

The additional benefit of this implementation is that by virtue of Rule 13.2.3: "Declare binary arithmetic and bitwise operators as non-members", these operators do not have to access to member data directly, however, they do not need to be declared as friends of the associated class.

Relational and Equality Operators

In principle `operator <` is sufficient to provide all other relational and equality operators.

For Example:

```
#include <utility>

class A
{
public:
  bool operator < (A const & rhs) const;

  bool operator == (A const & rhs) const
    { return !((*this) < rhs) && !(rhs < (*this)); }

  // Compliant
  bool operator != (A const & rhs) const
    { return std::rel_ops::operator != (*this, rhs); }
  bool operator <= (A const & rhs) const
    { return std::rel_ops::operator <= (*this, rhs); }
  bool operator > (A const & rhs) const
    { return std::rel_ops::operator > (*this, rhs); }
```

```
  bool operator >= (A const & rhs) const
    { return std::rel_ops::operator >= (*this, rhs); }
};
```

However, `operator ==` is not required to be defined as above, as a direct implementation may be more efficient, or when relational operators are not implemented for a particular class.

Increment and Decrement Operators

The post-increment operator should be implemented in terms of pre- increment. Similarly, for the decrement operators.

For Example:

```
#include <cstdint>

class A
{
public:
  A ();

  A& operator ++ ();  // pre-increment
  A operator ++ (int) // Compliant: post-increment
  {
    A result (*this);
    this->operator ++ ();
    return result;
  }

  A& operator -- ();  // pre-decrement
  A operator -- (int) // Non-Compliant: post-decrement
  {
    A result (*this);
    --m_i;
    return result;
  }

public:
  int32_t m_i;
};
```

References:

- HIC++ v3.3 – 3.1.9

14 Templates

14.1 Template declarations

14.1.1 Use variadic templates rather than an ellipsis

Use of the ellipsis notation `...` to indicate an unspecified number of arguments should be avoided. Variadic templates offer a type-safe alternative.

For Example:

```
#include <iostream>

int print ()
{
  std::cout << std::endl;
}

// Compliant: variadic template function
template<typename First, typename ... Rest>
void print (First const & v, Rest const & ... args)
{
  std::cout << v;
  print (args ...); // recursive template instantiation or a call to print ()
}

void foo (int32_t i)
{
  print ("And␣only␣", i, "␣little␣ducks␣came␣back.");
}
```

If use of a variadic template is not possible, function overloading or function call chaining (e.g. similar to stream output) should be considered.

References:

- HIC++ v3.3 – 11.6
- JSF AV C++ Rev C – 108
- MISRA C++:2008 – 8-4-1

14.2 Template instantiation and specialization

14.2.1 Declare template specializations in the same file as the primary template they specialize.

Partial and explicit specializations of function and class templates should be declared with the primary template. This will ensure that implicit specializations will only occur when there is no explicit declaration available.

For Example:

```
// file.h
template<typename T>
void foo (T & t)
```

```
{
  ++t;
}

// file1.cpp
#include "file.h"

void bar1 ()
{
  int32_t i = 3;
  foo<int32_t> (i);
  // primary template used, i is now 4
}

// file2.cpp
#include "file.h"

// Non-Compliant
template<>
void foo<int32_t> (int32_t & t)
{
  --t;
}

void bar2 ()
{
  int32_t i = 3;
  foo<int32_t> (i);
  // explicit specialization used, i is now 2
}
```

References:

- JSF AV C++ Rev C – 104
- MISRA C++:2008 – 14-7-3

14.2.2 Do not explicitly specialize a function template that is overloaded with other templates

Overload resolution does not take into account explicit specializations of function templates. Only after overload resolution has chosen a function template will any explicit specializations be considered.

For Example:

```
#include <cstdint>

template<typename T>
void f1 (T);                    // #1

template<typename T>
void f1 (T*);                   // #2

template<>
void f1<int32_t*> (int32_t*); // #3 // Non-Compliant (Explicit specialization of #1)
```

```
void f2 (int32_t * p)
{
  f1(p);                        // Calls #2
  f1<int32_t*>(p);              // Calls #3
}
```

References:

- MISRA C++:2008 – 14-8-1

14.2.3 Declare `extern` an explicitly instantiated template

Declaring the template with `extern` will disable implicit instantiation of the template when it is used in other translation units, saving time and reducing compile time dependencies.

For Example:

```
#include <cstdint>

// t.h
template <typename T>
class A1 { };

template <typename T>
class A2 { };

extern template class A2<int32_t>;

// t.cpp
#include "t.h"
template class A1<int32_t>;          // Non-Compliant
template class A2<int32_t>;          // Compliant
```

15 Exception handling

15.1 Throwing an exception

15.1.1 Only use instances of `std::exception` for exceptions

Exceptions pass information up the call stack to a point where error handling can be performed. If an object of class type is thrown, the class type itself serves to document the cause of an exception.

Only types that inherit from `std::exception`, should be thrown.

For Example:

```
#include <cstdint>
#include <stdexcept>
#include <iostream>

int foo ();

void bar ()
{
  try
  {
    if (0 == foo ())
    {
      throw -1;      // Non-Compliant
    }
  }
  catch (int32_t e) // Non-Compliant
  {
  }

  try
  {
    if (0 == foo ())
    {
      throw std::runtime_error ("unexpected condition"); // Compliant
    }
  }
  catch (std::exception const & e)                      // Compliant
  {
    std::cerr << e.what ();
  }
}
```

If an instance of an object inheriting from `std::exception` is created, then such an object must appear in a `throw` expression.

For Example:

```
#include <exception>

class MyException : public std::exception
```

```
{
  // ...
};

void f1 ()
{
  MyException myExcp;    // Non-Compliant
}

void f2 ()
{
  MyException myExcp;    // Compliant
  throw myExcp;
}
```

References:

- HIC++ v3.3 – 9.2

15.2 Constructors and destructors

15.2.1 Do not throw an exception from a destructor

The 2011 C++ Language Standard states that unless a user provided destructor has an explicit exception specification, one will be added implicitly, matching the one that an implicit destructor for the type would have received.

For Example:

```
#include <stdexcept>
class A
{
public:
  ~A ()          // Non-Compliant:  Implicit destructor for A would be declared with
                 // noexcept, therefore this destructor is noexcept
  {
    throw std::runtime_error ("results␣in␣call␣to␣std::terminate");
  }
};
```

Furthermore when an exception is thrown, stack unwinding will call the destructors of all objects with automatic storage duration still in scope up to the location where the exception is eventually caught.

The program will immediately terminate should another exception be thrown from a destructor of one of these objects.

For Example:

```
#include <cstdint>
#include <stdexcept>

class A
{
public:
  A () : m_p () {}
```

```
  ~A () noexcept(false)
  {
    if (nullptr == m_p)
    {
      throw std::runtime_error ("null␣pointer␣in␣A"); // Non-Compliant
    }
  }

private:
  int32_t * m_p;
};

void foo (int32_t i)
{
  if (i < 0)
  {
    throw std::range_error ("i␣is␣negative");
  }
}

void bar ()
{
  try
  {
    A a;

    foo (-1);
  }
  catch (std::exception const & e)
  {
  }
}
```

References:

- HIC++ v3.3 – 9.1

15.3 Handling an exception

15.3.1 Do not access non-static members from a `catch` handler of constructor/destructor *function try block*

When a constructor or a destructor has a function try block, accessing a non-static member from an associated exception handler will result in undefined behavior.

For Example:

```
#include <cstdint>

class C
{
public:
```

```
  C ();

private:
  int32_t m_i;
};

C::C ()
try : m_i()
{
  // constructor body
  ++m_i; // Compliant
}
catch (...)
{
  --m_i; // Non-Compliant
}
```

References:

- MISRA C++:2008 – 15-3-3

15.3.2 Ensure that a program does not result in a call to `std::terminate`

The path of an exception should be logical and well defined. Throwing an exception that is never subsequently caught, or attempting to rethrow when an exception is not being handled is an indicator of a problem with the design.

For Example:

```
bool f1 ();

void f2 ()
{
  throw;
}

void f3 ()
{
  try
  {
    if (f1 ())
    {
      throw float(0.0);
    }
    else
    {
      f2();                  // Non-Compliant: No current exception
    }
  }
  catch (...)
  {
    f2();
  }
}
```

```
int main ()
{
  f3();                          // Non-Compliant:  If 'float' thrown
}
```

References:

- MISRA C++:2008 – 15-3-2

16 Preprocessing

16.1 Source file inclusion

16.1.1 Use the preprocessor only for implementing include guards, and including header files with include guards

The preprocessor should only be used for including header files into other headers or the main source file, in order to form a translation unit. In particular only the following include directive forms should be used:

- `#include <xyz>`
- `#include "xyz"`

For Example:

```
// Compliant
#include <stddef>

// Non-Compliant
#define MYHEADER "stddef"
#include MYHEADER

// Non-Compliant
#define CPU 1044
#ifndef CPU
#error "no␣CPU␣defined"
#endif
```

Additionally, an include guard should be present in each header file, to prevent multiple inclusions of the same file. A header file can take one of the following forms:

For Example:

```
// only comments or whitespace
#ifndef UNIQUE_IDENT_IN_PROJECT
#define UNIQUE_IDENT_IN_PROJECT
// Compliant
// declarations
#endif
// only comments or whitespace
```

For Example:

```
// only comments or whitespace
#if ! defined (UNIQUE_IDENT_IN_PROJECT)
#define UNIQUE_IDENT_IN_PROJECT
// Compliant
// declarations
#endif
// only comments or whitespace
```

Where `UNIQUE_IDENT_IN_PROJECT` is chosen to uniquely represent the file which is being guarded.

Preprocessor macros do not obey the linkage, lookup and function call semantics. Instead use constant objects, inline functions and function templates.

For Example:

```
#include <cstdint>
#include <algorithm>

#define MIN(a,b) (((a) < (b)) ? (a) : (b))

void foo (int32_t i, int32_t j)
{
  int32_t k = MIN(i,j); // Non-Compliant
  k = std::min (i, j);  // Compliant
}
```

References:

- HIC++ v3.3 – 14.11
- JSF AV C++ Rev C – 26
- JSF AV C++ Rev C – 27
- JSF AV C++ Rev C – 28
- JSF AV C++ Rev C – 29
- JSF AV C++ Rev C – 30
- JSF AV C++ Rev C – 31
- MISRA C++:2008 – 16-2-1
- MISRA C++:2008 – 16-2-2

16.1.2 Do not include a path specifier in filenames supplied in `#include` directives

Hardcoding the path to a header file in a `#include` directive may necessitate changes to source code when it is reused or ported.

Alternatively, the directory containing the header file should be passed to the compiler on command line (e.g. `-i` or `/i` option).

For Example:

```
// Non-Compliant
#include "../../component/include/api.h"

// Non-Compliant:  may work on Windows
#include "..\\..\\component\\include\\api.h"

// Compliant
#include "api.h"
```

References:

- HIC++ v3.3 – 14.10
- MISRA C++:2008 – 16-2-5

16.1.3 Match the filename in a `#include` directive to the one on the filesystem

Some operating systems have case insensitive filesystems. Code initially developed on such a system may not compile successfully when ported to a case sensitive filesystem.

For Example:

```
// Non-Compliant
#include <CStdDef>

// Compliant
#include <cstddef>
```

References:

- HIC++ v3.3 – 14.12

16.1.4 Use <> brackets for system and standard library headers. Use quotes for all other headers

It is common practice that `#include <...>` is used for compiler provided headers, and `#include "..."` for user provided files.

Adhering to this guideline therefore helps with the understandability and maintainability of the code.

For Example:

```
// Non-Compliant
#include "cstddef"

// Compliant
#include <cstddef>

// Non-Compliant
#include <types.h>

// Compliant
#include "types.h"
```

References:

- HIC++ v3.3 – 14.9

16.1.5 Include directly the minimum number of headers required for compilation

Presence of spurious include directives can considerably slow down compilation of a large code base. When a source file is refactored, the list of included headers should be reviewed, to remove include directives which are no longer needed.

Doing so may also offer an opportunity to delete from code repository source files that are no longer used in the project, therefore reducing the level of technical debt.

For Example:

```
// Compliant
#include <cstddef>

// Non-Compliant: not used
#include <vector>
```

```
// Non-Compliant: duplicate
#include <cstddef>

void foo (std::size_t s);
```

To improve compilation time further, where possible, forward declaration of a user defined type should be preferred to including a header file that defines the type.

For Example:

```
class C; // Compliant
class D; // Compliant

C foo (D);

C * bar (D const &);
```

References:

- HIC++ v3.3 – 17.20

17 Standard library

17.1 General

17.1.1 Do not use `std::vector<bool>`

The `std::vector<bool>` specialization does not conform to the requirements of a container and does not work as expected in all STL algorithms.

In particular &v[0] does not return a contiguous array of elements as it does for other vector types. Additionally, the C++ Language Standard guarantees that different elements of an STL container can safely be modified concurrently, except for a container of `std::vector<bool>` type.

For Example:

```
#include <cstdint>
#include <vector>

void foo ()
{
  std::vector <int32_t> vi; // Compliant
  std::vector <bool> vb;    // Non-Compliant
}
```

References:

- HIC++ v3.3 – 17.13

17.2 The C standard library

17.2.1 Wrap use of the C Standard Library

The C11 standard library, which is included in the C++ standard library, leaves the handling of concerns relating to security and concurrency up to the developer.

Therefore, if the C standard library is to be used, it should be wrapped, with the wrappers ensuring that undefined behavior and data races will not occur.

For Example:

```
#include <cstdio>
#include <cerrno>

bool foo ()
{
  std::puts ("hello world"); // Non-Compliant
  return (0 == errno);       // Non-Compliant
}
```

The wrapper code should be placed in separate source files, and this rule should be deviated for those files only.

References:

- JSF AV C++ Rev C – 17
- MISRA C++:2008 – 19-3-1

17.3 General utilities library

17.3.1 Do not use `std::move` on objects declared with `const` or `const &` type

An object with `const` or `const &` type will never actually be moved as a result of calling `std::move`.

For Example:

```
#include <cstdint>
#include <utility>

template <typename T> void f1 (T&&);          //  #1
template <typename T> void f1 (T const &);    //  #2

void f2 (int32_t const & i)
{
  f1(i);                                      //  Calls #2
  f1(std::move(i));                           //  Non-Compliant:  Calls #1
}
```

17.3.2 Use `std::forward` to forward universal references

The `std::forward` function takes the value category of universal reference parameters into account when passing arguments through to callees.

When passing a non universal reference argument `std::move` should be used.

Note: As `auto` is implemented with argument deduction rules, an object declared with `auto &&` is also a universal reference for the purposes of this rule.

For Example:

```
#include <utility>
#include <cstdint>

template <typename ...T>
void f1 (T...t);

template <typename T1, typename T2>
void f2 (T1 && t1, T2 & t2)
{
  f1( std::forward<T1>(t1) );    // Compliant
  f1( std::forward<T2>(t2) );    // Non-Compliant
  f1( std::move(t1) );           // Non-Compliant
  f1( std::move(t2) );           // Compliant
}
void f3()
{
```

```
  int32_t i;
  f2(0, i);
}
```

17.3.3 Do not subsequently use the argument to `std::forward`

Depending on the value category of arguments used in the call of the function, `std::forward` may or may not result in a move of the parameter.

When the value category of the parameter is an *lvalue*, then modifications to the parameter will affect the argument of the caller. In the case of an *rvalue*, the value should be considered as being indeterminate after the call to `std::forward` (See Rule 8.4.1: "Do not access an invalid object or an object with indeterminate value").

For Example:

```
#include <cstdint>
#include <utility>

template <typename T1, typename T2>
void bar (T1 const & t1, T2 & t2);

template <typename T1, typename T2>
void foo (T1 && t1, T2 && t2)
{
  bar (std::forward<T1> (t1), std::forward<T2> (t2));
  ++t2;  // Non-Compliant
}

int main ()
{
  int32_t i = 0;
  foo (0, i);
}
```

17.3.4 Do not create smart pointers of array type

Memory allocated with array new must be deallocated with array delete. A smart pointer that refers to an array object must have this information passed in when the object is created. A consequence of this is that it is not possible to construct such a smart pointer using `std::make_shared`.

A `std::array` or `std::vector` can be used in place of the raw array type. The usage and performance will be very similar but will not have the additional complexity required when deallocating the array object.

For Example:

```
#include <memory>
#include <array>
#include <vector>
#include <cstdint>

typedef std::vector<int32_t> int_seq;

void foo ()
{
```

```
  // Non-Compliant
  std::unique_ptr<int32_t[]> oa_1 (new int32_t[10]);

  // Non-Compliant
  std::shared_ptr<int32_t> ob_1 (new int32_t[10]
    , std::default_delete<int32_t[]>());

  // Compliant
  std::array<int32_t, 10> oa_2;

  // Compliant
  std::shared_ptr< int_seq > ob_2 (std::make_shared<int_seq>( 10
    , int32_t() ));
}
```

References:

- 17.4.2: Use API calls that construct objects in place

17.3.5 Do not create an *rvalue reference* of `std::array`

The `std::array` class is a wrapper for a C style array. The cost of moving `std::array` is linear with each element of the array being moved. In most cases, passing the array by `&` or `const &` will provide the required semantics without this cost.

For Example:

```
#include <array>
#include <cstdint>

void f1(std::array<int32_t, 10> const &);    // Compliant
void f2(std::array<int32_t, 10> &&);         // Non-Compliant
```

17.4 Containers library

17.4.1 Use const container calls when result is immediately converted to a const iterator

The 2011 C++ Language Standard introduced named accessors for returning const iterators. Using these members removes an implicit conversion from `iterator` to `const_iterator`.

Another benefit is that the declaration of the iterator object can then be changed to use `auto` without the danger of affecting program semantics.

For Example:

```
#include <vector>
#include <cstdint>

void f(std::vector<int32_t> & v)
{
  // Non-Compliant
  for(std::vector<int32_t>::const_iterator iter = v.begin (), end = v.end ()
      ; iter != end
      ; ++iter)
```

```
  {
  }

  // Compliant
  for(std::vector<int32_t>::const_iterator iter = v.cbegin (), end = v.cend ()
      ; iter != end
      ; ++iter)
  {
  }

  // Compliant
  for(auto iter = v.cbegin (), end = v.cend ()
      ; iter != end
      ; ++iter)
  {
  }
}
```

17.4.2 Use API calls that construct objects in place

The 2011 C++ Language Standard allows for perfect forwarding. This allows for the arguments to a constructor to be passed through an API and therefore allowing for the final object to be constructed directly where it is intended to be used.

For Example:

```
#include <memory>
#include <cstdint>

void foo ()
{
  // Non-Compliant
  std::shared_ptr<int32_t> p1 = std::shared_ptr<int32_t>(new int32_t(0));

  // Compliant
  std::shared_ptr<int32_t> p2 = std::make_shared<int32_t>(0);
}
```

Analogous `make_unique` template is currently missing from the standard library, but one could easily be defined locally.

For Example:

```
#include <memory>
#include <utility>

namespace high_integrity
{
  // make_unique not yet available in C++ '11.  This
  // version from: http://herbsutter.com/gotw/_102/

  template <typename T, typename ...Args>
  std::unique_ptr<T> make_unique( Args&& ...args )
  {
```

```
    return std::unique_ptr<T>( new T( std::forward<Args>(args)... ) );
  }
}
using high_integrity::make_unique;
```

References:

- 17.3.4: Do not create smart pointers of array type
- Sutter Guru of the Week (GOTW) – 102

17.5 Algorithms Library

17.5.1 Do not ignore the result of `std::remove`, `std::remove_if` or `std::unique`

The mutating algorithms `std::remove`, `std::remove_if` and both overloads of `std::unique` operate by swapping or moving elements of the range they are operating over.

On completion, they return an iterator to the last valid element. In the majority of cases the correct behavior is to use this result as the first operand in a call to `std::erase`.

For Example:

```
#include <vector>
#include <algorithm>
#include <iostream>
#include <cstdint>

int main ()
{
  std::vector<int32_t> v1 = { 0, 0, 1, 1, 2, 2, 3, 3, 4, 4 };
  std::unique(v1.begin(), v1.end ());                          // Non-Compliant

  // The possible contents of the vector are:
  // { 0, 1, 2, 3, 4, 2, 3, 3, 4, 4 };

  std::vector<int32_t> v2 = { 0, 0, 1, 1, 2, 2, 3, 3, 4, 4 };
  v2.erase (std::unique(v2.begin(), v2.end ()), v2.end ()); // Compliant
}
```

References:

- MISRA C++:2008 – 0-1-7

18 Concurrency

18.1 General

18.1.1 Do not use platform specific multi-threading facilities

Rather than using platform-specific facilities, the C++ standard library should be used as it is platform independent.

For Example:

```
// Non-Compliant
#include <pthread.h>
void* thread1(void*);
void f1()
{
  pthread_t t1;
  pthread_create(&t1, nullptr, thread1, 0);
  // ...
}

// Compliant
#include <thread>
void thread2();
void f2()
{
  std::thread t1(thread2);
  // ...
}
```

References:

- Williams Concurrency – 1.3.4
- Meyers Effective C++ '11 (draft TOC) – Use native handles to transcend the C++11 API

18.2 Threads

18.2.1 Use `high_integrity::thread` in place of `std::thread`

The destructor of `std::thread` will call `std::terminate` if the thread owned by the class is still joinable. By using a wrapper class a default behavior can be provided.

For Example:

```
// high_integrity.h
#include <thread>
#include <cstdint>

namespace high_integrity
{
  enum ThreadExec : int32_t
  {
    DETACH,
```

```
    JOIN,
  };

  template <ThreadExec thread_exec>
  class thread
  {
  public:
    template <class F, class ...Args>
    thread (F&& f, Args&&...args)
      : m_thread(std::forward<F> (f), std::forward<Args>(args)...)
    {
    }
    thread(thread const &) = delete;
    thread(thread &&) = default;

    ~thread()
    {
      if(m_thread.joinable())
      {
        join_or_detach ();
      }
    }

    inline void join () { m_thread.join (); }

  private:
    inline void join_or_detach ();

  private:
    std::thread m_thread;
  };
  template <> void thread<ThreadExec::DETACH>::join_or_detach ()
  {
    m_thread.detach ();
  }

  template <> void thread<ThreadExec::JOIN>::join_or_detach ()
  {
    m_thread.join ();
  }
}

using high_integrity::thread;
using high_integrity::ThreadExec;

void f(int32_t);
int32_t main()
{
  int32_t i;

  // Non-Compliant:  Potentially calls 'std::terminate'
  std::thread t(f, i);
```

```
  // Compliant:  Will detach if required
  thread<ThreadExec::DETACH> hi_t (f, i);
}
```

References:

- Williams Concurrency – 2.1.3
- Meyers Effective C++ '11 (draft TOC) – Make std::threads unjoinable on all paths

18.2.2 Synchronize access to data shared between threads using a single lock

Using the same lock when accessing shared data makes it easier to verify the absence of problematic race conditions.

To help achieve this goal, access to data should be encapsulated such that it is not possible to read or write to the variable without acquiring the appropriate lock. This will also help limit the amount of code executed in the scope of the lock.

Note: Data may be referenced by more than one variable, therefore this requirement applies to the complete set of variables that could refer to the data.

For Example:

```
#include <mutex>
#include <string>
#include <cstdint>

class some_data
{
public:
  void do_something();

private:
  int32_t a;
  std::string b;
};

some_data* unprotected;

void malicious_function(some_data& protected_data)
{
  // Suspicious, unprotected now refers to data protected by a mutex
  unprotected=&protected_data;
}

class data_wrapper
{
public:
  template<typename Function>
  void process_data(Function func)
  {
    std::lock_guard<std::mutex> lk(m);
    func(data);          // 'protected_data' assigned to 'unprotected' here
```

```
  }

private:
  some_data data;
  mutable std::mutex m;
};

data_wrapper x;

void foo()
{
  x.process_data(malicious_function);

  // Not Compliant: 'unprotected' accessed outside of 'data_wrapper::m'
  unprotected->do_something();
}
```

Special attention needs to be made for const objects. The standard library expects operations on const objects to be thread-safe. Failing to ensure that this expectation is fulfilled may lead to problematic data races and undefined behavior. Therefore, operations on const objects of user defined types should consist of either reads entirely or internally synchronized writes.

For Example:

```
#include <mutex>
#include <atomic>
#include <cstdint>
#include "high_integrity.h"

class A
{
public:
  int32_t get1() const
  {
    ++counter1;     // Non-Compliant: unsynchronized write to a data
                    // member of non atomic type

    ++counter2;     // Compliant: write to a data member of atomic type
  }
  int32_t get2() const
  {
    std::lock_guard<std::mutex> guard(mut);
    ++counter1;     // Compliant: synchronized write to data member of non atomic type
  }
private:
  mutable std::mutex           mut;
  mutable int32_t              counter1;
  mutable std::atomic<int32_t> counter2;
};

using high_integrity::thread;
using high_integrity::ThreadExec;

void worker(A & a);
```

```
void foo(A & a)
{
  thread<ThreadExec::JOIN> thread (worker, std::ref (a));
}
```

References:

- Sutter Guru of the Week (GOTW) – 6a
- Williams Concurrency – 3.2.2
- Williams Concurrency – 3.2.8

18.2.3 Do not share volatile data between threads

Declaring a variable with the `volatile` keyword does not provide any of the required synchronization guarantees:

- atomicity
- visibility
- ordering

For Example:

```
#include <functional>
#include <cstdint>
#include <unistd.h>
#include "high_integrity.h"

// Non-Compliant - using volatile for synchronization
class DataWrapper
{
public:
  DataWrapper ()
  : flag (false)
  , data (0)
  {
  }

  void incrementData()
  {
    while(flag)
    {
      sleep(1000);
    }
    flag = true;
    ++data;
    flag = false;
  }

  int32_t getData() const
  {
    while(flag)
    {
      sleep(1000);
```

```
    }
    flag = true;
    int32_t result (data);
    flag = false;

    return result;
  }

private:
  mutable volatile bool flag;
  int32_t data;
};

using high_integrity::thread;
using high_integrity::ThreadExec;

void worker(DataWrapper & data);
void foo(DataWrapper & data)
{
  thread<ThreadExec::JOIN> t (worker, std::ref (data));
}
```

Use mutex locks or ordered atomic variables, to safely communicate between threads and to prevent the compiler from optimizing the code incorrectly.

For Example:

```
#include <functional>
#include <cstdint>
#include <mutex>
#include "high_integrity.h"

// Compliant - using locks
class DataWrapper
{
public:
  DataWrapper ()
  : data (0)
  {
  }

  void incrementData()
  {
    std::lock_guard<std::mutex> guard(mut);
    ++data;
  }

  int32_t getData() const
  {
    std::lock_guard<std::mutex> guard(mut);
    return data;
  }

private:
```

```
  mutable std::mutex mut;
  int32_t data;
};

using high_integrity::thread;
using high_integrity::ThreadExec;

void worker(DataWrapper & data);
void foo(DataWrapper & data)
{
  thread<ThreadExec::JOIN> t (worker, std::ref (data));
}
```

References:

- CERT C++ – CON01-CPP
- Meyers Effective C++ '11 (draft TOC) – Distinguish volatile from std::atomic<>
- Sutter Concurrency – 19

18.2.4 Use `std::call_once` rather than the Double-Checked Locking pattern

The Double-Checked Locking pattern can be used to correctly synchronize initializations.

For Example:

```
#include <memory>
#include <atomic>
#include <mutex>
#include <cstdint>

std::mutex mut;
static std::atomic<int32_t *> instance;

int & getInstance ()
{
  // Non-Compliant:  Using double-checked locking pattern
  if (! instance.load (std::memory_order_acquire))
  {
    std::lock_guard<std::mutex> lock (mut);
    if (!instance.load (std::memory_order_acquire))
    {
      int32_t * i = new int32_t (0);
      instance.store (i, std::memory_order_release);
    }
  }

  return * instance.load (std::memory_order_relaxed);
}
```

However, the C++ standard library provides `std::call_once` which allows for a cleaner implementation:

For Example:

```
#include <mutex>
```

```
#include <cstdint>

int32_t * instance;
std::once_flag initFlag;

void init ()
{
  instance = new int32_t (0);
}

int32_t & getInstance ()
{
  // Compliant: Using 'call_once'
  std::call_once (initFlag, init);
  return *instance;
}
```

Initialization of a local object with static storage duration is guaranteed by the C++ Language Standard to be re-entrant. However this conflicts with Rule 3.3.1: "Do not use variables with static storage duration", which takes precedence.

For Example:

```
#include <cstdint>

int32_t & getInstance ()
{
  // Non-Compliant: using a local static
  static int32_t instance (0);
  return instance;
}
```

References:

- Williams Concurrency – 3.3.1

18.3 Mutual Exclusion

18.3.1 Within the scope of a lock, ensure that no static path results in a lock of the same mutex

It is undefined behavior if a thread tries to lock a `std::mutex` it already owns, this should therefore be avoided.

For Example:

```
#include <mutex>
#include <cstdint>

std::mutex mut;
int32_t i;

void f2(int32_t j);

void f1(int32_t j)
{
```

```
  std::lock_guard<std::mutex> hold(mut);
  if (j)
  {
    f2(j);
  }
  ++i;
}

void f2(int32_t j)
{
  if (! j)
  {
    std::lock_guard<std::mutex> hold(mut); // Non-Compliant:  "Static Path" Exists
                                           // to here from f1
    ++i;
  }
}
```

References:

- Williams Concurrency – 3.3.3

18.3.2 Ensure that order of nesting of locks in a project forms a DAG

Mutex locks are a common causes of deadlocks. Multiple threads trying to acquire the same lock but in a different order may end up blocking each other.

When each lock operation is treated as a vertex, two consecutive vertices with no intervening lock operation in the source code are considered to be connected by a directed edge. The resulting graph should have no cycles, i.e. it should be a Directed Acyclic Graph (DAG).

For Example:

```
#include <cstdint>
#include <mutex>

// Non-Compliant:  Nesting of locks does not form a DAG: mut1->mut2 and then mut2->mut1
class A
{
public:
  void f1()
  {
    std::lock_guard<std::mutex> lock1(mut1);
    std::lock_guard<std::mutex> lock2(mut2);
    ++i;
  }

  void f2()
  {
    std::lock_guard<std::mutex> lock2(mut2);
    std::lock_guard<std::mutex> lock1(mut1);
    ++i;
  }
```

```
private:
  std::mutex mut1;
  std::mutex mut2;
  int32_t i;
};

// Compliant:  Nesting of locks forms a DAG: mut1->mut2 and then mut1->mut2
class B
{
public:
  void f1()
  {
    std::lock_guard<std::mutex> lock1(mut1);
    std::lock_guard<std::mutex> lock2(mut2);
    ++i;
  }

  void f2()
  {
    std::lock_guard<std::mutex> lock1(mut1);
    std::lock_guard<std::mutex> lock2(mut2);
    ++i;
  }

private:
  std::mutex mut1;
  std::mutex mut2;
  int32_t i;
};
```

References:

- Williams Concurrency – 3.2.4
- Williams Concurrency – 3.2.5
- Sutter Concurrency – 5
- Sutter Concurrency – 6

18.3.3 Do not use `std::recursive_mutex`

Use of `std::recursive_mutex` is indicative of bad design: Some functionality is expecting the state to be consistent which may not be a correct assumption since the mutex protecting a resource is already locked.

For Example:

```
// Non-Compliant:  Using recursive_mutex
#include <mutex>
#include <cstdint>

class DataWrapper
{
public:
  int32_t incrementAndReturnData()
```

```
  {
    std::lock_guard<std::recursive_mutex> guard(mut);
    incrementData();
    return data;
  }

  void incrementData()
  {
    std::lock_guard<std::recursive_mutex> guard(mut);
    ++data;
  }

  // ...
private:
  mutable std::recursive_mutex mut;
  int32_t data;
};
```

Such situations should be solved by redesigning the code.

For Example:

```
// Compliant:  Not using mutex
#include <mutex>
#include <cstdint>

class DataWrapper
{
public:
  int32_t incrementAndReturnData()
  {
    std::lock_guard<std::mutex> guard(mut);
    inc();
    return data;
  }

  void incrementData()
  {
    std::lock_guard<std::mutex> guard(mut);
    inc();
  }

  // ...
private:
  void inc()
  {
    // expects that the mutex has already been locked
    ++data;
  }

  mutable std::mutex mut;
  int32_t data;
};
```

References:

- Williams Concurrency – 3.3.3

18.3.4 Only use `std::unique_lock` when `std::lock_guard` cannot be used

The `std::unique_lock` type provides additional features not available in `std::lock_guard`. There is an additional cost when using `std::unique_lock` and so it should only be used if the additional functionality is required.

For Example:

```
#include <functional>
#include <mutex>
#include <cstdint>

std::unique_lock<std::mutex> getGlobalLock ();

void f1(int32_t val)
{
  static std::mutex mut;
  std::unique_lock<std::mutex> lock(mut); // Non-Compliant
  // ...
}

void f2()
{
  auto lock = getGlobalLock ();            // Compliant
  // ...
}
```

References:

- Williams Concurrency – 3.2.6

18.3.5 Do not access the members of `std::mutex` directly

A mutex object should only be managed by the `std::lock_guard` or `std::unique_lock` object that owns it.

For Example:

```
#include <mutex>

std::mutex mut;
void f()
{
  std::lock_guard<std::mutex> lock(mut);
  mut.unlock ();                          // Non-Compliant
}
```

References:

- 3.4.3: Use RAII for resources

18.3.6 Do not use relaxed atomics

Using non-sequentially consistent memory ordering for atomics allows the CPU to reorder memory operations resulting in a lack of total ordering of events across threads. This makes it extremely difficult to reason about the correctness of the code.

For Example:

```
#include <atomic>
#include <cstdint>

template<typename T>
class CountingConsumer
{
public:
  explicit CountingConsumer(T *ptr, int32_t counter)
    : m_ptr(ptr), m_counter(counter)
  { }

  void consume (int data)
  {
    m_ptr->consume (data);

    // Non-Compliant
    if (m_counter.fetch_sub (1, std::memory_order_release) == 1)
    {
      delete m_ptr;
    }
  }

  T * m_ptr;
  std::atomic<int32_t> m_counter;
};
```

References:

- Sutter Hardware – Part 2

18.4 Condition Variables

18.4.1 Do not use `std::condition_variable_any` on a `std::mutex`

When using `std::condition_variable_any`, there is potential for additional costs in terms of size, performance or operating system resources, because it is more general than `std::condition_variable`.

`std::condition_variable` works with `std::unique_lock<std::mutex>`, while `std::condition_variable_any` can operate on any objects that have `lock` and `unlock` member functions.

For Example:

```
#include <mutex>
#include <condition_variable>
#include <vector>
#include <cstdint>

std::mutex mut;
```

```
std::condition_variable_any cv;
std::vector<int32_t> container;

void producerThread()
{
  int32_t i = 0;
  std::lock_guard<std::mutex> guard(mut);

  // critical section
  container.push_back(i);

  cv.notify_one();
}

void consumerThread()
{
  std::unique_lock<std::mutex> guard(mut);

  // Non-Compliant:  conditional_variable_any used with std::mutex based lock 'guard'
  cv.wait(guard, []{ return !container.empty(); } );

  // critical section
  container.pop_back();
}
```

References:

- Williams Concurrency – 4.1.1

References

CERT C++	CERT C++ Secure Coding Standard, https://www.securecoding.cert.org
C++11	The ISO C++ Language Standard ISO/IEC 14882:2011, http://www.open-std.org/jtc1/sc22/wg21/
C++98	The ISO C++ Language Standard ISO/IEC 14882:1998, http://www.open-std.org/jtc1/sc22/wg21/
Going Native 2013	Stephan T. Lavavej: Going Native 2013: Don't Help the Compiler, http://channel9.msdn.com
HIC++ v3.3	High Integrity C++ Coding Standard Manual - Version 3.3, September 2012, Programming Research
JSF AV C++ Rev C	Joint Strike Fighter Air Vehicle C++ Coding Standards, December 2005, Lockheed Martin Corporation
Meyers Effective C++ '11 (draft TOC)	Scott Meyers: Draft TOC for Effective C++11 Concurrency Chapter, http://scottmeyers.blogspot.hu
Meyers Notes	Scott Meyers: Overview of The New C++ (C++11/14), July 2013, http://www.artima.com
MISRA C++:2008	MISRA C++:2008 Guidelines for the use of the C++ language in critical systems, June 2008, MIRA Limited
Sutter Hardware	Herb Sutter: The C++ Memory Model and Modern Hardware, http://herbsutter.com
Sutter Concurrency	Herb Sutter: Effective Concurrency Columns, August 2007 - September 2010, Dr. Dobb's Journal, http://www.gotw.ca/publications
Sutter Guru of the Week (GOTW)	Herb Sutter: Guru of the Week, http://herbsutter.com/gotw/
Williams Concurrency	Anthony Williams: C++ Concurrency in Action - Practical Multithreading, 2012, Manning Publications Co.

High Integrity 3.3 to 4.0 Rule Mappings

Old	New	Rule Text in HIC++ 4.0	Change
1.3.1	1.1.1	Ensure that code complies with the 2011 ISO C++ Language Standard	merged/revised for C++11
3.1.3	12.5.1	Define explicitly `=default` or `=delete` implicit special member functions of concrete classes	merged/revised for C++11
3.1.4	12.5.6	Use an atomic, non-throwing swap operation to implement the copy and move assignment operators	revised for C++11
3.1.5	12.5.7	Declare assignment operators with the ref-qualifier `&`	revised for C++11
3.1.8	9.1.1	Declare `static` any member function that does not require `this`. Alternatively, declare `const` any member function that does not modify the externally visible state of the object	extended
3.1.9	13.2.5	Implement a minimal set of operators and use them to implement all other related operators	extended
3.1.10	12.1.1	Do not declare implicit user defined conversions	merged
3.1.11	12.1.1	Do not declare implicit user defined conversions	merged
3.1.13	12.5.1	Define explicitly `=default` or `=delete` implicit special member functions of concrete classes	merged/revised for C++11
3.2.1	12.4.2	Ensure that a constructor initializes explicitly all base classes and non-static data members	revised for C++11
3.2.2	12.4.4	Write members in an initialization list in the order in which they are declared	identical
3.2.3	12.1.1	Do not declare implicit user defined conversions	merged/revised for C++11
3.2.5	3.4.3	Use RAII for resources	merged
3.3.2	12.2.1	Declare `virtual`, `private` or `protected` the destructor of a type used as a base class	relaxed
3.3.3	5.4.3	Do not convert from a base class to a derived class	reworded
3.3.5	13.1.1	Ensure that all overloads of a function are visible from where it is called	merged
3.3.11	13.1.1	Ensure that all overloads of a function are visible from where it is called	merged
3.3.12	9.1.2	Make default arguments the same or absent when overriding a virtual function	reworded
3.3.13	12.4.1	Do not use the dynamic type of an object unless the object is fully constructed	extended
3.3.14	12.5.8	Make the copy assignment operator of an abstract class protected or define it `=delete`	revised for C++11
3.3.15	10.1.1	Ensure that access to base class subobjects does not require explicit disambiguation	reworded
3.3.16	10.2.1	Use the `override` special identifier when overriding a virtual function	revised for C++11
3.4.1	11.1.1	Declare all data members private	reworded
3.4.2	9.1.3	Do not return non-const handles to class data from const member functions	identical
3.4.3	9.1.4	Do not write member functions which return non-const handles to data less accessible than the member function	reworded
3.4.4	11.2.1	Do not use friend declarations	reworded
3.4.6	10.3.1	Ensure that a derived class has at most one base class which is not an interface class	extended

continued on next page

continued from previous page

Old	New	Rule Text in HIC++ 4.0	Change
3.5.1	13.2.1	Do not overload operators with special semantics	extended
3.5.3	13.2.2	Ensure that the return type of an overloaded binary operator matches the built-in counterparts	relaxed
3.5.4	13.2.3	Declare binary arithmetic and bitwise operators as non-members	relaxed
3.5.5	13.2.4	When overloading the subscript operator (`operator[]`) implement both const and non-const versions	identical
4.1	8.3.1	Do not write functions with an excessive McCabe Cyclomatic Complexity	identical
4.2	8.3.2	Do not write functions with a high static program path count	reworded
4.3	8.2.2	Do not declare functions with an excessive number of parameters	reworded
5.1	6.1.1	Enclose the body of a selection or an iteration statement in a compound statement	reworded
5.2	4.2.2	Ensure that data loss does not demonstrably occur in an integral expression	merged
5.3	1.2.1	Ensure that all statements are reachable	extended
5.4	6.1.3	Ensure that a non-empty case statement block does not fall through to the next label	reworded
5.5	6.2.4	Only modify a for loop counter in the for expression	reworded
5.6	6.2.3	Do not alter a control or counter variable more than once in a loop	reworded
5.8	6.3.1	Ensure that the label(s) for a jump statement or a switch condition appear later, in the same or an enclosing block	relaxed
5.10	6.3.2	Ensure that execution of a function with a non-void return type ends in a return statement with a value	reworded
5.11	6.1.2	Explicitly cover all paths through multi-way selection statements	reworded
5.12	6.4.1	Postpone variable definitions as long as possible	merged
6.1	4.2.1	Ensure that the `U` suffix is applied to a literal used in a context requiring an unsigned integral expression	reworded
6.2	4.3.1	Do not convert an expression of wider floating point type to a narrower floating point type	merged
6.4	1.1.1	Ensure that code complies with the 2011 ISO C++ Language Standard	merged
6.5	2.5.1	Do not concatenate strings with different encoding prefixes	revised for C++11
7.1	5.4.1	Only use casting forms: `static_cast` (excl. `void*`), `dynamic_cast` or explicit constructor call	merged
7.3	5.4.1	Only use casting forms: `static_cast` (excl. `void*`), `dynamic_cast` or explicit constructor call	merged
7.4	5.4.1	Only use casting forms: `static_cast` (excl. `void*`), `dynamic_cast` or explicit constructor call	merged
7.5	5.4.1	Only use casting forms: `static_cast` (excl. `void*`), `dynamic_cast` or explicit constructor call	merged
7.6	4.4.1	Do not convert floating values to integral types except through use of standard library functions	identical
7.7	5.4.1	Only use casting forms: `static_cast` (excl. `void*`), `dynamic_cast` or explicit constructor call	merged
8.1.1	7.4.3	Ensure that an object or a function used from multiple translation units is declared in a single header file	reworded

continued on next page

continued from previous page

Old	New	Rule Text in HIC++ 4.0	Change
8.1.2	7.4.2	Ensure that an inline function, a function template, or a type used from multiple translation units is defined in a single header file	merged
8.1.3	7.4.2	Ensure that an inline function, a function template, or a type used from multiple translation units is defined in a single header file	merged
8.2.1	3.1.1	Do not hide declarations	reworded
8.2.2	3.3.1	Do not use variables with static storage duration	reworded
8.2.3	7.3.1	Do not use *using directives*	reworded
8.3.1	7.4.1	Ensure that any objects, functions or types to be used from a single translation unit are defined in an unnamed namespace in the main source file	reworded
8.3.3	1.3.2	Do not use the `register` keyword	revised for C++11
8.3.4	2.4.1	Ensure that each identifier is distinct from any other visible identifier	identical
8.4.2	7.1.1	Declare each identifier on a separate line in a separate declaration	merged
8.4.3	8.4.1	Do not access an invalid object or an object with indeterminate value	revised for C++11
8.4.4	6.4.1	Postpone variable definitions as long as possible	merged
8.4.5	7.1.6	Use class types or typedefs to abstract scalar quantities and standard integer types	merged
8.4.6	7.1.6	Use class types or typedefs to abstract scalar quantities and standard integer types	merged
8.4.7	7.1.1	Declare each identifier on a separate line in a separate declaration	merged
8.4.11	7.1.2	Use `const` whenever possible	identical
8.4.13	4.2.2	Ensure that data loss does not demonstrably occur in an integral expression	merged/relaxed
9.1	15.2.1	Do not throw an exception from a destructor	identical
9.2	15.1.1	Only use instances of `std::exception` for exceptions	extended
9.5	3.4.3	Use RAII for resources	merged
10.1	5.1.1	Use symbolic names instead of literal values in code	identical
10.2	5.2.1	Ensure that pointer or array access is demonstrably within bounds of a valid object	reworded
10.3	5.1.2	Do not rely on the sequence of evaluation within an expression	merged
10.4	5.1.3	Use parentheses in expressions to specify the intent of the expression	identical
10.5	5.1.2	Do not rely on the sequence of evaluation within an expression	merged
10.7	4.2.2	Ensure that data loss does not demonstrably occur in an integral expression	merged/relaxed
10.9	5.1.6	Do not code side effects into the right-hand operands of: `&&`, `||`, `sizeof`, `typeid` or a function passed to `condition_variable::wait`	revised for C++11
10.10	1.2.2	Ensure that no expression or sub-expression is redundant	extended
10.11	5.6.1	Do not use bitwise operators with signed operands	reworded
10.12	4.2.2	Ensure that data loss does not demonstrably occur in an integral expression	merged

continued on next page

continued from previous page

Old	New	Rule Text in HIC++ 4.0	Change
10.13	4.2.2	Ensure that data loss does not demonstrably occur in an integral expression	merged/relaxed
10.14	4.3.1	Do not convert an expression of wider floating point type to a narrower floating point type	merged/relaxed
10.15	5.7.1	Do not write code that expects floating point calculations to yield exact results	identical
10.16	1.3.1	Do not use the increment operator (`++`) on a variable of type `bool`	identical
10.17	5.5.1	Ensure that the right hand operand of the division or remainder operators is demonstrably non-zero	reworded
10.19	5.1.2	Do not rely on the sequence of evaluation within an expression	merged
10.20	5.8.1	Do not use the conditional operator (`?:`) as a sub-expression	relaxed
10.21	5.3.1	Do not apply unary minus to operands of unsigned type	reworded
11.3	8.2.1	Make parameter names absent or identical in all declarations	relaxed
11.4	8.2.3	Pass small objects with a trivial copy constructor by value	merged
11.5	8.2.3	Pass small objects with a trivial copy constructor by value	merged
11.6	14.1.1	Use variadic templates rather than an ellipsis	revised for C++11
11.7	3.4.1	Do not return a reference or a pointer to an automatic variable defined within the function	reworded
11.8	7.1.5	Do not inline large functions	extended
12.2	5.3.2	Allocate memory using `new` and release it using `delete`	identical
12.3	5.3.3	Ensure that the form of `delete` matches the form of `new` used to allocate the memory	reworded
12.5	3.4.3	Use RAII for resources	merged
12.6	12.3.1	Correctly declare overloads for `operator new` and `delete`	merged
12.7	12.3.1	Correctly declare overloads for `operator new` and `delete`	merged
12.8	3.4.3	Use RAII for resources	merged/relaxed
13.3	1.1.1	Ensure that code complies with the 2011 ISO C++ Language Standard	merged
13.5	7.5.1	Do not use the `asm` declaration	identical
13.6	3.5.1	Do not make any assumptions about the internal representation of a value or object	merged
13.7	5.4.1	Only use casting forms: `static_cast` (excl. `void*`), `dynamic_cast` or explicit constructor call	merged
14.1	2.3.1	Do not use the C comment delimiters `/* ... */`	identical
14.2	2.1.1	Do not use tab characters in source files	identical
14.9	16.1.4	Use `<>` brackets for system and standard library headers. Use quotes for all other headers	identical
14.10	16.1.2	Do not include a path specifier in filenames supplied in `#include` directives	identical
14.11	16.1.1	Use the preprocessor only for implementing include guards, and including header files with include guards	extended
14.12	16.1.3	Match the filename in a `#include` directive to the one on the filesystem	relaxed
14.16	2.5.3	Use `nullptr` for the null pointer constant	revised for C++11
14.18	2.2.1	Do not use digraphs or trigraphs	identical
15.1	3.5.1	Do not make any assumptions about the internal representation of a value or object	merged
15.4	5.4.2	Do not cast an expression to an enumeration type	reworded

continued on next page

continued from previous page

Old	New	Rule Text in HIC++ 4.0	Change
17.1	1.3.3	Do not use the C Standard Library .h headers	reworded
17.13	17.1.1	Do not use `std::vector<bool>`	identical
17.20	16.1.5	Include directly the minimum number of headers required for compilation	extended
17.21	1.3.4	Do not use deprecated STL library features	extended

Revision History

Issue	Date
1.0	3 October 2003
2.0	20 October 2003
2.1	24 October 2003
2.2	18 May 2004
2.3	6 October 2006
2.4	14 December 2006
3.0	24 January 2008
3.1	15 February 2008
3.2	3 October 2008
3.3	25 September 2012
4.0	3 October 2013

Conditions of Use