Document Title	Guidelines for the use of the C++14 language in critical and safety-related systems
Document Owner	AUTOSAR
Document Responsibility	AUTOSAR
Document Identification No	839

Document Status	Final
Part of AUTOSAR Standard	Adaptive Platform
Part of Standard Release	18-10

Document Change History

Date	Release	Changed by	Description
2018-10-31	18-10	AUTOSAR Release Management	• Added traceability for ISO 26262 (B.6) • New rules resulting from continued analysis of the C++ Core Guideline • Finished addressing MISRA review comments of the 2017-03 release • Improvements of already existing rules, more details in the Changelog (D.3) • Marked the specification as obsolete
2018-03-29	18-03	AUTOSAR Release Management	• New rules resulting from the analysis of JSF, HIC, CERT, C++ Core Guideline • Improvements of already existing rules, more details in the Changelog (D.2) • Covered smart pointers usage • Reworked checked/unchecked exception definitions and rules
2017-10-27	17-10	AUTOSAR Release Management	• Updated traceability for HIC, CERT, C++ Core Guideline • Partially included MISRA review of the 2017-03 release • Changes and fixes for existing rules, more details in the Changelog (D.1)

2017-03-31	17-03	AUTOSAR Release Management	• Initial release

Disclaimer

Contents

1 Background

See chapter 1. Background" in MISRA C++:2008, which is applicable for this document as well.

This document specifies coding guidelines for the usage of the C++14 language as defined by ISO/IEC 14882:2014 [3], in the safety-related and critical systems. The main application sector is automotive, but it can be used in other embedded application sectors.

This document is defined as an update of MISRA C++:2008 [7]. The rules that are adopted from MISRA C++ without modifications, are only referred in this document by ID and rule text, without repeating their complete contents. Therefore, MISRA C++:2008 is required prerequisite for the readers of this document. MISRA C++:2008 can be purchased over MISRA web store. The reference to the adopted MISRA C++:2008 rules is not considered as a reproduction of a part of MISRA C++:2008.

Most of the rules are automatically enforceable by static analysis. Some are partially enforceable or even non-enforceable and they need to be enforced by a manual code review.

Most of the rules are typical coding guidelines i.e. how to write code. However, for the sake of completeness and due to the fact that some rules are relaxed with respect to MISRA C++:2008 (e.g. exceptions and dynamic memory is allowed), there are also some rules related to compiler toolchain and process-related rules concerning e.g. analysis or testing.

This document is not about the style of code in a sense of naming conventions, layout or indentation. But as there are several C++ code examples, they need some form of style guide convention. Therefore, the code examples are written in a similar way like the MISRA C++:2008 code examples.

2 The vision

2.1 Rationale for the production of AUTOSAR C++14

Currently, no appropriate coding standards for C++14 or C++11 exist for the use in critical and safety-related software. Existing standards are incomplete, covering old C++ versions or not applicable for critical/safety-related. In particular, MISRA C++:2008 does not cover C++11/14. Therefore this document is to cover this gap.

MISRA C++:2008 is covering the C++03 language, which is 13 years old at the time of writing this document. In the meantime, the market evolved, by:

1. substantial evolution/improvement of C++ language
2. more widespread use of object-oriented languages in safety-related and critical environments
3. availability of better compilers
4. availability of better testing, verification and analysis tools appropriate for C++
5. availability of better development methodologies (e.g. continuous integration) that allow to detect/handle errors earlier
6. higher acceptance of object-oriented languages by safety engineers and
7. strong needs of development teams for a powerful C++ language features
8. creation of ISO 26262 safety standard, which HIC++, JSF++, CERT C++, C++ Core Guidelines

As a result, MISRA C++:2008 requires an update. This document is therefore an add-on on MISRA and it specifies:

1. which MISRA rules are obsolete and do not need to be followed
2. a number of updated MISRA rules (for rules that only needed some improvements)
3. several additional rules.

Moreover, at the time of writing, MISRA C++:2008 was already not complete / fully appropriate. For example, it completely disallows dynamic memory, standard libraries are not fully covered, security is not covered.

2.2 Objectives of AUTOSAR C++14

This document specifies coding guidelines for the usage of the C++14 language, in the safety-related and critical environments, as an update of MISRA C++:2008, based on other leading coding standards and the research/analysis done by AUTOSAR. The

main application sector is automotive, but it can be used in other embedded application sectors.

The AUTOSAR C++14 Coding Guidelines addresses high-end embedded micro-controllers that provide efficient and full C++14 language support, on 32 and 64 bit micro-controllers, using POSIX or similar operating systems.

For the ISO 26262 clauses allocated to software architecture, unit design and implementation, the document provides an interpretation of how these clauses apply specifically to C++.

3 Scope

See also chapter "3. Scope" in MISRA C++:2008, which is applicable for this document as well.

This document specifies coding guidelines for the usage of the C++14 language as defined by ISO/IEC 14882:2014 [3], in the safety-related and critical environments, as an update of MISRA C++:2008. The main application sector is automotive, but it can be used in other embedded application sectors.

The document is built using the MISRA C++:2008 document structure, document logic and convention and formatting. Each rule is specified using the MISRA C++:2008 pattern and style.

Several rules from MISRA C++:2008 were adopted without modifications. See B.1 for the comparison. The adopted MISRA rules are only referenced by ID and title, without providing the full contents. The inclusion of ID and of the rule title for the adopted rules is considered not be a "reproduction".

Several other coding standards and resources are referenced in this document or used as a basis of the rules in this document:

1. Joint Strike Fighter Air Vehicle C++ Coding Standards [8]
2. High Integrity C++ Coding Standard Version 4.0 [9]
3. CERT C++ Coding Standard [10]
4. C++ Core Guidelines [11]
5. Google C++ Style Guide [12]

3.1 Allowed features of C++ language

This document allows most of C++ language features, but with detailed restrictions, as expressed by the rules. This has an important impact on the compiler toolchains, as well as other software development tools, as these tools need to provide a full support of the C++ features (as long as these features are used in accordance to the coding guidelines).

The document allows in particular the usage of dynamic memory, exceptions, templates, inheritance and virtual functions. On the other side, the compiler toolchain needs to provide them correctly. In most cases, this requires a tool qualification.

The explanatory summary table 3.1 lists features introduced in C++11 and C++14 and it also summarizes pre-C++11 features, together with their support by the coding standard.

Category:	Feature:	Since:	May be used:	Shall not be used:
6.0 Language independent issues				
	Dynamic memory management	-	X	
	Floating-point arithmetic	-	X	
6.1 General				
	Operators new and delete	-	X	
	malloc and free functions	-		X
	Sized deallocation	C++11	X	
6.2 Lexical conventions				
	Namespaces	-	X	
6.3 Basic Concepts				
	Fixed width integer types	C++11	X	
6.4 Standard Conversions				
	Nullptr pointer literal	C++11	X	
6.5 Expressions				
	C-style casts	-		X
	const_cast conversion	-		X
	dynamic_cast conversion	-		X
	reinterpret_cast conversion	-		X
	static_cast conversion	-	X	
	Lambda expressions	C++11	X	
	Binary literals	C++14	X	
6.6 Statements				
	Range-based for loops	C++11	X	
	goto statement	-		X
6.7 Declaration				
	constexpr specifier	C++11	X	
	auto specifier	C++11	X	
	decltype specifier	C++11	X	
	Generic lambda expressions	C++14	X	
	Trailing return type syntax	C++11	X	
	Return type deduction	C++14		X
	typedef specifier	-		X
	using specifier	C++11	X	
	Scoped enumerations	C++11	X	
	std::initializer_list	C++11	X	
	asm declaration	-		X
6.8 Declarators				
	Default arguments	-	X	
	Variadic arguments	-		X

	List initialization	C++11	X	
6.9 Classes				
	Unions	-		X
	Bit-fields	-	X	
6.10 Derived Classes				
	Inheritance	-	X	
	Multiple inheritance	-		X
	Virtual functions	-	X	
	override specifier	C++11	X	
	final specifier	C++11	X	
6.11 Member Access Control				
	friend declaration	-		X
6.12 Special Member Functions				
	Defaulted and deleted functions	C++11	X	
	Delegating constructors	C++11	X	
	Member initializer lists	-	X	
	Non-static data member initializer	C++11	X	
	explicit specifier	-	X	
	Move semantics	C++11	X	
6.13 Overloading				
	User-defined literals	C++11	X	
	Digit sequences separators ’	C++14	X	
6.14 Templates				
	Variadic templates	C++11	X	
	Variable templates	C++14	X	
6.15 Exception Handling				
	Exceptions	-	X	
	Function-try-blocks	-		X
	Dynamic exception specification	-		X
	noexcept specifier	C++11	X	
6.16 Preprocessing Directives				
	Static assertion	C++11	X	
	Implementation defined behavior control (#pragma directive)	-		X

Table 3.1: C++14 features

3.2 Limitations

In the current release, the following are known limitations:

1. The rule set for parallel computing is not provided

2. The rule set for security (as long as it is not common to critical software or safety-related software) is not provided
3. The rule set for C++ standard libraries is partial (incomplete)
4. All remaining non-analyzed rules from CERT and HIC++ are concurrency/security related
5. The traceability to C++ Core Guidelines contains some non-analyzed rules

The limitations will be addressed in future versions of this document.

If the user of this document uses parallel computing, C++ standard libraries or develops security-related software, then they are responsible to apply their own guidelines for these topics.

Further analysis of the following rules will be made for a future release: A2-10-1, A5-1-8, A7-1-3, A7-1-5, A12-1-1, A12-1-2, A12-1-3, A12-4-1, A12-4-2, A13-5-3, A14-7-1, A16-0-1, A16-2-2, A16-7-1, A17-1-1, A18-9-2, A27-0-2, A27-0-4.

4 Using AUTOSAR C++14

See chapter "4. Using MISRA C++" in MISRA C++:2008, which is applicable for this document as well.

5 Introduction to the rules

5.1 Rule classification

5.1.1 Rule classification according to compatibility with MISRA

The rules in this document are defined as a “delta” to MISRA C++:2008. Therefore, the rules are of two types from this perspective:

5.1.2 Rule classification according to obligation level

The rules are classified according to obligation level:

- required: These are mandatory requirements placed on the code. C++ code that is claimed to conform to AUTOSAR C++14 shall comply with every “Required” rule. Formal deviations must be raised where this is not the case.
- advisory: These are requirements placed on the code that should normally be followed. However they do not have the mandatory status of “Required” rules. Note that the status of “Advisory” does not mean that these items can be ignored, but that they should be followed as far as is reasonably practical. Formal deviations are not necessary for “Advisory” rules, but may be raised if it is considered appropriate.

5.1.3 Rule classification according to enforcement by static analysis

The rules are classified according to enforcement by static code analysis tools:

- automated: These are rules that are automatically enforceable by means of static analysis.
- partially automated: These are the rules that can be supported by static code analysis, e.g. by heuristic or by covering some error scenarios, as a support for a manual code review.
- non-automated: These are the rules where the static analysis cannot provide any reasonable support by a static code analysis and they require other means, e.g. manual code review or other tools.

Most of the rules are automatically enforceable by a static analysis. A static code analysis tool that claims a full compliance to this standard shall fully check all “enforceable static analysis” rules and it shall check the rules that are “partially enforceable by static analysis” to the extent that is possible/reasonable.

The compliance to all rules that are not “enforceable by static analysis” shall be ensured by means of manual activities like review, analyses.

5.1.4 Rule classification according to allocated target

Finally, the rules are classified according to the target:

- implementation: These are the rules that apply to the implementation of the project (code and to software design and architecture).
- verification: These are the rules that apply to the verification activities (e.g. code review, analysis, testing).
- toolchain: These are the rules that apply to the toolchain (preprocessor, compiler, linker, compiler libraries).
- infrastructure: These are the rules that apply to the operating system and the hardware.

5.2 Organization of rules

The rules are organized in chapter 6, similar to the structure of ISO/IEC 14882:2014 document. In addition, rules that do not fit to this structure are defined in chapter 6.0.

5.3 Exceptions to the rules

Some rules contain an Exception section that lists one or more exceptional conditions under which the rule need not be followed. These exceptions effectively modify the headline rule.

5.4 Redundancy in the rules

There are a few cases within this document where rules are partially overlapping (redundant). This is intentional.

Firstly, this approach brings often more clarity and completeness. Secondly, it is because several redundant rules are reused from MISRA C++:2008. Third, it may be that the developer chooses to raise a deviation against one of the partially overlapping rules, but not against others.

For example, goto statement is prohibited by rule A6-6-1 and the usage of goto is restricted by rules M6-6-1 and M6-6-2 that are overlapping to A6-6-1. So if the developer decides to deviate from A6-6-1, they can still comply to M6-6-1 and M6-6-2.

5.5 Presentation of rules

The individual rules are presented in the format similar to the format of MISRA C++:2008.

5.6 Understanding the issue references

In this document release, references to C++ Language Standard are not provided.

5.7 Scope of rules

While the majority of rules can be applied within a single translation unit, all rules shall be applied with the widest possible interpretation.

In general, the intent is that all the rules shall be applied to templates. However, some rules are only meaningful for instantiated templates.

Unless otherwise specified, all rules shall apply to implicitly-declared or implicitly-defined special member functions (e.g. default constructor, copy constructor, copy assignment operator and destructor).

6 AUTOSAR C++14 coding rules

This chapter contains the specification of AUTOSAR C++14 coding rules.

6.0 Language independent issues

6.0.1 Unnecessary constructs

Rule M0-1-1 (required, implementation, automated)
A project shall not contain unreachable code.

See MISRA C++ 2008 [7]

Rule M0-1-2 (required, implementation, automated)
A project shall not contain infeasible paths.

See MISRA C++ 2008 [7]

Note: A path can also be infeasible because of a call to constexpr function which returned value, known statically, will never fulfill the condition of a condition statement.

Rule M0-1-3 (required, implementation, automated)
A project shall not contain unused variables.

See MISRA C++ 2008 [7]

Rule M0-1-4 (required, implementation, automated)
A project shall not contain non-volatile POD variables having only one use.

See MISRA C++ 2008 [7]

Rule A0-1-1 (required, implementation, automated)
A project shall not contain instances of non-volatile variables being given values that are not subsequently used.

Rationale

Known as a DU dataflow anomaly, this is a process whereby there is a data flow in which a variable is given a value that is not subsequently used. At best this is inefficient,

but may indicate a genuine problem. Often the presence of these constructs is due to the wrong choice of statement aggregates such as loops.

See: DU-Anomaly.

Exception

Loop control variables (see Section 6.6.5) are exempt from this rule.

Example

```
//% $Id: A0-1-1.cpp 289436 2017-10-04 10:45:23Z michal.szczepankiewicz $
#include <array>
#include <cstdint>
std::uint8_t Fn1(std::uint8_t param) noexcept
{
    std::int32_t x{
        0};  // Non-compliant - DU data flow anomaly; Variable defined,
             // but not used
    if (param > 0)
    {
        return 1;
    }
    else
    {
        return 0;
    }
}
std::int32_t Fn2() noexcept
{
    std::int8_t x{10U};  // Compliant - variable defined and will be used
    std::int8_t y{20U};  // Compliant - variable defined and will be used
    std::int16_t result = x + y;  // x and y variables used

    x = 0;  // Non-compliant - DU data flow anomaly; Variable defined, but x is
            // not subsequently used and goes out of scope
    y = 0;  // Non-compliant - DU data flow anomaly; Variable defined, but y is
            // not subsequently used and goes out of scope
    return result;
}
std::int32_t Fn3(std::int32_t param) noexcept
{
    std::int32_t x{param +
                   1};  // Compliant - variable defined, and will be used in
                        // one of the branches
                        // However, scope of x variable could be reduced
    if (param > 20)
    {
        return x;
    }
    return 0;
}
std::int32_t Fn4(std::int32_t param) noexcept
```

```
{
    std::int32_t x{param +
                   1};  // Compliant - variable defined, and will be used in
                        // some of the branches
    if (param > 20)
    {
        return x + 1;
    }
    else if (param > 10)
    {
        return x;
    }
    else
    {
        return 0;
    }
}
void Fn5() noexcept
{
    std::array<std::int32_t, 100> arr{};
    arr.fill(1);

    constexpr std::uint8_t limit{100U};
    std::int8_t x{0};
    for (std::uint8_t i{0U}; i < limit; ++i)  // Compliant by exception - on the
    // final loop, value of i defined will
    // not be used
    {
        arr[i] = arr[x];
        ++x;  // Non-compliant - DU data flow anomaly on the final loop, value
              // defined and not used
    }
}
```

See also

- MISRAC++2008: 0-1-6 A project shall not contain instances of non-volatile variables being given values that are never subsequently used.

Rule A0-1-2 (required, implementation, automated)
The value returned by a function having a non-void return type that is not an overloaded operator shall be used.

Rationale

A called function may provide essential information about its process status and result through return statement. Calling a function without using the return value should be a warning that incorrect assumptions about the process were made.

Overloaded operators are excluded, as they should behave in the same way as built-in operators.

Exception

The return value of a function call may be discarded by use of a static_cast<void> cast, so intentions of a programmer are explicitly stated.

Example

```
// $Id: A0-1-2.cpp 289436 2017-10-04 10:45:23Z michal.szczepankiewicz $
#include <algorithm>
#include <cstdint>
#include <vector>
std::uint8_t Fn1() noexcept
{
    return 0U;
}
void Fn2() noexcept
{
    std::uint8_t x = Fn1();     // Compliant
    Fn1();                      // Non-compliant
    static_cast<void>(Fn1());   // Compliant by exception
}
void Fn3()
{
    std::vector<std::int8_t> v{0, 0, 1, 1, 2, 2, 3, 3, 4, 4, 5, 5};
    std::unique(v.begin(), v.end());                      // Non-compliant
    v.erase(std::unique(v.begin(), v.end()), v.end());  // Compliant
}
```

See also

- MISRA C++ 2008 [7]: Rule 0-1-7 The value returned by a function having a non-void return type that is not an overloaded operator shall always be used.
- HIC++ v4.0 [9]: 17.5.1 Do not ignore the result of std::remove, std::remove_if or std::unique.

Rule M0-1-8 (required, implementation, automated)
All functions with void return type shall have external side effect(s).

See MISRA C++ 2008 [7]

Rule M0-1-9 (required, implementation, automated)
There shall be no dead code.

See MISRA C++ 2008 [7]

See also

- JSF December 2005 [8]: AV Rule 181: Redundant explicit casts will not be used.

Rule M0-1-10 (advisory, implementation, automated)
Every defined function should be called at least once.

See MISRA C++ 2008 [7]

Note: This rule enforces developers to statically and explicitly use every function in the source code. A function does not necessarily need to be called at run-time. Rule M0-1-1 detects all unreachable code occurrences.

Rule A0-1-3 (required, implementation, automated)
Every function defined in an anonymous namespace, or static function with internal linkage, or private member function shall be used.

Rationale

Functions which are not callable from outside the compilation unit in which they are defined, or from outside the class implementation to which they pertain, and which are not used may be symptomatic of serious problems, such as poor software design or missing paths in flow control.

This rule enforces developers to statically and explicitly use every such function in the source code. A function does not necessarily need to be called at run-time. Rule M0-1-1 detects all unreachable code occurrences.

Note that this rule applies equally to static and non-static private member functions.

Example

```
1  //% $Id: A0-1-3.cpp 291350 2017-10-17 14:31:34Z jan.babst $
2  #include <cstdint>
3  static void F1()  // Compliant
4  {
5  }
6
7  namespace
8  {
9    void F2()  // Non-compliant, defined function never used
10   {
11   }
12 }
13
14 class C
15 {
16   public:
17     C() : x(0) {}
18     void M1(std::int32_t i)  // Compliant, member function is used
19     {
20         x = i;
21     }
```

```
22      void M2(std::int32_t i,
23              std::int32_t j)  // Compliant, never used but declared
24                               // as public
25      {
26          x = (i > j) ? i : j;
27      }
28
29    protected:
30      void M1ProtectedImpl(std::int32_t j)  // Compliant, never used but declared
31                                            // as protected
32      {
33          x = j;
34      }
35
36    private:
37      std::int32_t x;
38      void M1PrivateImpl(
39          std::int32_t j)  // Non-compliant, private member function never used
40      {
41          x = j;
42      }
43  };
44  int main(int, char**)
45  {
46      F1();
47      C c;
48      c.M1(1);
49      return 0;
50  }
```

See also

- MISRA C++ 2008 [7]: Rule 0-1-10 Every defined function shall be called at least once.
- HIC++ v4.0 [9]: 1.2.2 Ensure that no expression or sub-expression is redundant.

Rule A0-1-4 (required, implementation, automated)
There shall be no unused named parameters in non-virtual functions.

Rationale

Unused named parameters are often a result of a design changes and can lead to mismatched parameter lists.

Note: This rule does not apply to unnamed parameters, as they are widely used in SFINAE and concept compliance.

Example

```
1  //% $Id: A0-1-4.cpp 305588 2018-01-29 11:07:35Z michal.szczepankiewicz $
2
3  #include <type_traits>
4  #include <string>
5
6  //Logger.hpp
7  class Logger
8  {
9  public:
10     struct console_t {};
11     struct file_t {};
12
13     constexpr static console_t console = console_t();
14     constexpr static file_t file = file_t();
15
16     void init(console_t);
17     void init(file_t, const std::string& prefix);
18 };
19
20 //Logger.cpp
21 void Logger::init(console_t)
22 {
23 //initialization for a console logger
24 }
25 void Logger::init(file_t, const std::string& prefix)
26 {
27 //initialization for a file logger for a given prefix path
28 }
29
30 //Message.h
31 struct MessagePolicy {};
32 struct WriteMessagePolicy final : public MessagePolicy { };
33
34 template <typename T> struct is_mutable : std::false_type {};
35 template <> struct is_mutable<WriteMessagePolicy> : std::true_type {};
36
37 template <typename T, typename Policy = MessagePolicy>
38 class Message
39 {
40 public:
41     static_assert(std::is_base_of<MessagePolicy, Policy>::value == true, "Given
    parameter is not derived from MessagePolicy");
42     using value_type = T;
43
44     template<typename U = void>
45     void set(T&& u, typename std::enable_if<is_mutable<Policy>::value, U>::type*
    = 0)
46     {
47         v = u;
48     }
49
```

```
private:
    value_type v;
};

int main(int, char**)
{
    Logger log;
    log.init(Logger::console);
    log.init(Logger::file, std::string("/tmp/"));

    Message<uint8_t> read;
    Message<uint8_t, WriteMessagePolicy> write;

    //read.set(uint8_t(12)); Compilation error
    write.set(uint8_t(12));

    return 0;
}
```

See also

- C++ Core Guidelines [11]: F.9: Unused parameters should be unnamed

Rule A0-1-5 (required, implementation, automated)
There shall be no unused named parameters in the set of parameters for a virtual function and all the functions that override it.

Rationale

Unused named parameters are often a result of a design changes and can lead to mismatched parameter lists.

Note: This rule does not apply to unnamed parameters, as overridden methods for some subclasses may need additional parameters.

Example

```
//% $Id: A0-1-5.cpp 305588 2018-01-29 11:07:35Z michal.szczepankiewicz $

#include <cstdint>
#include <vector>

//Compressor.h
class Compressor
{
public:
    using raw_memory_type = std::vector<uint8_t>;

    raw_memory_type Compress(const raw_memory_type& in, uint8_t ratio);

private:
```

```
15      virtual raw_memory_type __Compress(const raw_memory_type& in, uint8_t ratio)
        = 0;
16  };
17
18  //Compressor.cpp
19  Compressor::raw_memory_type Compressor::Compress(const raw_memory_type& in,
        uint8_t ratio)
20  {
21      return __Compress(in, ratio);
22  }
23
24  //JPEGCompressor.h
25  class JPEGCompressor : public Compressor
26  {
27  private:
28      raw_memory_type __Compress(const raw_memory_type& in, uint8_t ratio) override
        ;
29  };
30
31  //JPEGCompressor.cpp
32  JPEGCompressor::raw_memory_type JPEGCompressor::__Compress(const raw_memory_type&
        in, uint8_t ratio)
33  {
34      raw_memory_type ret;
35      //jpeg compression, ratio used
36      return ret;
37  }
38
39  //HuffmanCompressor.h
40  class HuffmanCompressor : public Compressor
41  {
42  private:
43      raw_memory_type __Compress(const raw_memory_type& in, uint8_t) override;
44  };
45
46  //JPEGCompressor.cpp
47  HuffmanCompressor::raw_memory_type HuffmanCompressor::__Compress(const
        raw_memory_type& in, uint8_t)
48  {
49      raw_memory_type ret;
50      //Huffman compression, no ratio parameter available in the algorithm
51      return ret;
52  }
```

See also

- C++ Core Guidelines [11]: F.9: Unused parameters should be unnamed

Rule A0-1-6 (advisory, implementation, automated)
There should be no unused type declarations.

Rationale

Unused type declarations make code unnecessary more complex and complicate review process. Unused types can be redundant or be unused by mistake.

Note: Libraries development require introduction new types not used internally.

Example

```
1 // $Id: A0-1-6.cpp$
2 #include <cstdint>
3
4 std::uint32_t Fn() noexcept
5 {
6     using LocalUIntPtr = std::uint32_t*;
7     return 0U;
8 }
```

See also

- MISRA C++ 2008 [7]: Rule 0-1-5 reclassified from required to advisory.

6.0.2 Storage

Rule M0-2-1 (required, implementation, automated)
An object shall not be assigned to an overlapping object.

See MISRA C++ 2008 [7]

6.0.3 Runtime failures

Rule M0-3-1 (required, implementation / verification, non-automated)
Minimization of run-time failures shall be ensured by the use of at least one of: (a) static analysis tools/techniques; (b) dynamic analysis tools/techniques; (c) explicit coding of checks to handle run-time faults.

See MISRA C++ 2008 [7]

Rule M0-3-2 (required, implementation, non-automated)
If a function generates error information, then that error information shall be tested.

See MISRA C++ 2008 [7]

Note: This rule does not cover exceptions due to different behavior. Exception handling is described in chapter 6.15.

6.0.4 Arithmetic

Rule M0-4-1 (required, implementation, non-automated)
Use of scaled-integer or fixed-point arithmetic shall be documented.

See MISRA C++ 2008 [7]

Rule M0-4-2 (required, implementation, non-automated)
Use of floating-point arithmetic shall be documented.

See MISRA C++ 2008 [7]

Rule A0-4-1 (required, infrastructure / toolchain, non-automated)
Floating-point implementation shall comply with IEEE 754 standard.

Rationale

Floating-point arithmetic has a range of problems associated with it. Some of these can be overcome by using an implementation that conforms to IEEE 754 (IEEE Standard for Floating-Point Arithmetic).

Note that the rule implies that toolchain, hardware, C++ Standard Library and C++ built-in types (i.e. float, double) will provide full compliance to IEEE 754 standard in order to use floating-points in the project.

Also, see: A0-4-2.

Example

```
1 //% $Id: A0-4-1.cpp 271389 2017-03-21 14:41:05Z piotr.tanski $
2 #include <limits>
3 static_assert(
4     std::numeric_limits<float>::is_iec559,
5     "Type float does not comply with IEEE 754 single precision format");
6 static_assert(
```

```
    std::numeric_limits<float>::digits == 24,
    "Type float does not comply with IEEE 754 single precision format");

static_assert(
    std::numeric_limits<double>::is_iec559,
    "type double does not comply with IEEE 754 double precision format");
static_assert(
    std::numeric_limits<double>::digits == 53,
    "Type double does not comply with IEEE 754 double precision format");
```

See also

- MISRA C++ 2008 [7]: Rule 0-4-3 Floating-point implementations shall comply with a defined floating-point standard.
- JSF December 2005 [8]: AV Rule 146 Floating point implementations shall comply with a defined floating point standard.

Rule A0-4-2 (required, implementation, automated)
Type long double shall not be used.

Rationale

The width of long double type, and therefore width of the significand, is implementation-defined.

The width of long double type can be either:

- 64 bits, as the C++14 Language Standard allows long double to provide at least as much precision as type double does, or
- 80 bits, as the IEEE 754 standard allows extended precision formats (see: Extended-Precision-Format), or
- 128 bits, as the IEEE 754 standard defines quadruple precision format

Example

```
//% $Id: A0-4-2.cpp 289436 2017-10-04 10:45:23Z michal.szczepankiewicz $
void Fn() noexcept
{
    float f1{0.1F};         // Compliant
    double f2{0.1};         // Compliant
    long double f3{0.1L};   // Non-compliant
}
```

See also

- none

Rule A0-4-3 (required, toolchain, automated)
The implementations in the chosen compiler shall strictly comply with the C++14 Language Standard.

Rationale

It is important to determine whether implementations provided by the chosen compiler strictly follow the ISO/IEC 14882:2014 C++ Language Standard.

Example

Since the ISO/IEC 14882:2014 C++ Language Standard, the integer division and modulo operator results are no longer implementation-defined. The sentence "if both operands are nonnegative then the remainder is nonnegative; if not, the sign of the remainder is implementation-defined" from ISO/IEC 14882:2003 is no longer present in the standard since ISO/IEC 14882:2011. Note that this rule also covers the modulo operator as it is defined in terms of integer division.

Deducing the type of an auto variable initialized using `auto x{<value>}` is implemented differently depending on the language standard. In C++11 and C++14, `x` will be a `std::initializer_list`, whereas in C++17, `x` will be a type deduced from the specified `<value>`. Furthermore, some compilers may already implement the C++17 behavior even when operated in C++14 mode.

Note: Rule A8-5-3 forbids initializing an `auto` variable with the curly braces (`{}`) syntax.

Other features provided by the chosen compiler also should follow the ISO/IEC 14882:2014 C++ Language Standard.

See also

- MISRA C++ 2008 [7]: Rule 1-0-3 The implementation of integer division in the chosen compiler shall be determined and documented.
- C++ Core Guidelines [11]: F.46: int is the return type for main().

Rule A0-4-4 (required, implementation, partially automated)
Range, domain and pole errors shall be checked when using math functions.

Rationale

The C Standard defines the following types of error related to math functions specifically:

- domain error – input arguments are outside a domain of a mathematical function definition
- pole error – for finite input arguments a function gives an exact infinite result

- range error – a result of a mathematical function cannot be represented by the return type limitations

Domain and pole errors require that bounds are checked for input parameters before calling a mathematical function. Range errors in most cases cannot be prevented, as their occurrence mostly depend on the implementation of floating-point numbers (see A0-4-1).

Checking for range errors for multi-threaded applications require that floating-point exception state is in a per-thread basis.

Example

```
//% $Id: A0-4-4.cpp 305588 2018-01-29 11:07:35Z michal.szczepankiewicz $

#include <cmath>
#include <cfenv>

float Foo(float val)
{
    //non-compliant, domain error for negative values
    return std::sqrt(val);
}

float Bar(float val)
{
    //non-compliant
    //domain error for val < 0
    //pole error for val==0
   return std::log(val);
}

// \return true, if a range error occurred
bool DetectRangeErr()
{
    return ((math_errhandling & MATH_ERREXCEPT) &&
        (fetestexcept(FE_INEXACT | FE_OVERFLOW | FE_UNDERFLOW) != 0));
}
```

See also

- SEI CERT C++ Coding Standard [10]: FLP32-C: Prevent or detect domain and range errors in math functions

6.1 General

6.1.1 Scope

Rule A1-1-1 (required, implementation, automated)
All code shall conform to ISO/IEC 14882:2014 - Programming Language C++ and shall not use deprecated features.

Rationale

The current version of the C++ language is as defined by the ISO International Standard ISO/IEC 14822:2014(E) "Information technology - Programming languages - C++".

The C++14 is the improved version of the C++11. It is also “the state of the art” of C++ development that is required by ISO 26262 standard [6].

Any reference in this document to “C++ Language Standard” refers to the ISO/IEC 14822:2014 standard.

Note that all of the deprecated features of C++ Language Standard are defined in ISO/IEC 14882:2014 - Programming Language C++ Annexes C “Compatibility” and D “Compatibility features”.

Example

```
1  //% $Id: A1-1-1.cpp 289436 2017-10-04 10:45:23Z michal.szczepankiewicz $
2  #include <cstdint>
3  #include <stdexcept>
4  void F(std::int32_t i)
5  {
6      std::int32_t* a = nullptr;
7
8      // __try  // Non-compliant - __try is a part of Visual Studio extension
9      try  // Compliant - try keyword is a part of C++ Language Standard
10     {
11         a = new std::int32_t[i];
12         // ...
13     }
14
15     // __finally  // Non-compliant - __finally is a part of Visual Studio
16     // extension
17     catch (
18         std::exception&)  // Compliant - C++ Language Standard does not define
19                           // finally block, only try and catch blocks
20     {
21         delete[] a;
22         a = nullptr;
23     }
24 }
```

See also

- MISRA C++ 2008 [7]: 1-0-1 All code shall conform to ISO/IEC 14882:2003 “The C++ Standard Incorporating Technical Corrigendum 1”

- JSF December 2005 [8]: 4.4.1 All code shall conform to ISO/IEC 14882:2002(E) standard C++.
- HIC++ v4.0 [9]: 1.1.1 Ensure that code complies with the 2011 ISO C++ Language Standard.
- HIC++ v4.0 [9]: 1.3.4 Do not use deprecated STL library features.

Rule M1-0-2 (required, toolchain, non-automated)
Multiple compilers shall only be used if they have a common, defined interface.

See MISRA C++ 2008 [7]

Rule A1-1-2 (required, implementation / toolchain, non-automated)
A warning level of the compilation process shall be set in compliance with project policies.

Rationale

If compiler enables the high warning level, then it is able to generate useful warning messages that point out potential run-time problems during compilation time. The information can be used to resolve certain errors before they occur at run-time.

Note that it is common practice to turn warnings into errors.

Also, note that enabling the highest compiler warning level may produce numerous useless messages during compile time. It is important that the valid warning level for the specific compiler is established in the project.

See also

- JSF December 2005 [8]: AV Rule 218 Compiler warning levels will be set in compliance with project policies.

Rule A1-1-3 (required, toolchain, non-automated)
An optimization option that disregards strict standard compliance shall not be turned on in the chosen compiler.

Rationale

Enabling optimizations that disregard compliance with the C++ Language Standard may create an output program that should strictly comply to the standard no longer valid.

See also

- none

6.1.2 Normative references

Rule A1-2-1 (required, toolchain, non-automated)
When using a compiler toolchain (including preprocessor, compiler itself, linker, C++ standard libraries) in safety-related software, the tool confidence level (TCL) shall be determined. In case of TCL2 or TCL3, the compiler shall undergo a "Qualification of a software tool", as per ISO 26262-8.11.4.6 [6].

Rationale

Vulnerabilities and errors in the compiler toolchain impact the binary that is built.

Example

The following mechanisms could help to increase the Tool error Detection (TD) and thus allowing to reduce the Tool Confidence Level:

1. Achievement of MC/DC code coverage on generated project assembly code
2. Diverse implementation of safety requirements at software or even at system level (e.g. two micro-controllers)
3. Usage of diverse compilers or compilation options
4. Diversity at the level of operating system
5. Extensive testing (e.g. equivalence class testing, boundary value testing), testing at several levels (e.g. unit testing, integration testing)

Note that in most automotive applications, the compiler is evaluated TCL3 or TCL2. In case of TCL2 or TCL3, the following are typically performed (by compiler vendor or by a project), see table 4 in ISO 26262-8:

1. Evaluation of the tool development process
2. Validation of the software tool, by performing automatic compiler tests that are derived from the C++ language specification

See also

- ISO 26262-8 [6]: 11 Confidence in the use of software tools.

6.1.4 Implementation compliance

Rule A1-4-1 (required, implementation / verification, non-automated)
Code metrics and their valid boundaries shall be defined and code shall comply with defined boundaries of code metrics.

Rationale

Code metrics that concern i.e. project's structure, function's complexity and size of a source code shall be defined at the project level. It is also important to determine valid boundaries for each metric to define objectives of the measurement.

Source code metrics needs to be measured for the project and comply with defined boundaries. This gives valuable information whether the source code is complex, maintainable and efficient.

See also

- HIC++ v4.0 [9]: 8.3.1 Do not write functions with an excessive McCabe Cyclomatic Complexity.
- HIC++ v4.0 [9]: 8.3.2 Do not write functions with a high static program path count.
- HIC++ v4.0 [9]: 8.2.2 Do not declare functions with an excessive number of parameters.

Rule A1-4-3 (advisory, implementation, automated)
All code should compile free of compiler warnings.

Rationale

Compiler warnings provide the earliest tool-supported indication of potential problems in source code. Developers should not ignore compiler warnings.

Note: Compiler warnings should be turned on to a level matching (as far as possible) the rules in this document, in particular A1-1-1. A possible enforcement of this rule is to turn compiler warnings into errors.

See also

- A1-1-1 in section 6.1.1

6.2 Lexical conventions

6.2.3 Character sets

Rule A2-3-1 (required, architecture / design / implementation, automated)
Only those characters specified in the C++ Language Standard basic source character set shall be used in the source code.

Rationale

"The basic source character set consists of 96 characters: the space character, the control characters representing horizontal tab, vertical tab, form feed, and new-line,

plus the following 91 graphical characters: a b c d e f g h i j k l m n o p q r s t u v w x y z A B C D E F G H I J K L M N O P Q R S T U V W X Y Z 0 1 2 3 4 5 6 7 8 9 _ { } [ ] # ( ) < > % : ; . ? * + - / ^ & | ~ ! =, \ " ’

” [C++ Language Standard [3]]

Exception

It is permitted to use other characters inside the text of a wide string and a UTF-8 encoded string literal.

It is also permitted to use a character @ inside comments. See rule A2-7-3.

Example

```
// $Id: A2-3-1.cpp 307578 2018-02-14 14:46:20Z michal.szczepankiewicz $
#include <cstdint>

void Fn() noexcept
{
    std::int32_t sum = 0;  // Compliant
    // std::int32_t Â£_value = 10;  // Non-compliant
    // sum += Â£_value;  // Non-compliant
    // Variable sum stores Â£ pounds  // Non-compliant
}
```

See also

- JSF December 2005 [8]: AV Rule 9: Only those characters specified in the C++ basic source character set will be used.

6.2.5 Alternative tokens

Rule A2-5-1 (required, implementation, automated)
Trigraphs shall not be used.

Rationale

Trigraphs are denoted to be a sequence of 2 question marks followed by a specified third character (e.g. ??’ represents a ~character. They can cause accidental confusion with other uses of two question marks.

The Trigraphs are: ??=, ??/, ??’, ??(, ??), ??!, ??<, ??>, ??-.

Example

```
//% $Id: A2-5-1.cpp 289436 2017-10-04 10:45:23Z michal.szczepankiewicz $
#include <iostream>
void Fn1()
{
```

```
5       std::cout << "Enter date ??/??/??";  // Non-compliant, ??/??/?? becomes \\??
6                                             // after trigraph translation
7   }
8   void Fn2()
9   {
10      std::cout << "Enter date dd/mm/yy";  // Compliant
11  }
```

See also

- MISRA C++2008: Rule 2-3-1 (Required) Trigraphs shall not be used.
- JSF December 2005 [8]: AV Rule 11 Trigraphs will not be used.
- HIC++ v4.0 [9]: 2.2.1 Do not use digraphs or trigraphs.

Rule A2-5-2 (required, implementation, automated)
Digraphs shall not be used.

Rationale

The digraphs are: <%, %>, <:, :>, %:, %:%:.

The use of digraphs may not meet developer expectations.

Example

```
1   //% $Id: A2-5-2.cpp 305382 2018-01-26 06:32:15Z michal.szczepankiewicz $
2   class A
3   {
4     public:
5       void F2() {}
6   };
7   // void fn1(A* a<:10:>)  // Non-compliant
8   // <%
9   //   a<:0:>->f2();
10  // %>
11  void Fn2(A* a[10])  // Compliant, equivalent to the above
12  {
13      a[0]->F2();
14  }
```

See also

- MISRA C++ 2008 [7]: advisory 2-5-1 Digraphs should not be used.
- JSF December 2005 [8]: 4.4.1 AV Rule 12 The following digraphs will not be used.
- HIC++ v4.0 [9]: 2.2.1 Do not use digraphs or trigraphs.

6.2.7 Comments

Rule M2-7-1 (required, implementation, automated)
The character sequence /* shall not be used within a C-style comment.

See MISRA C++ 2008 [7]

Rule A2-7-1 (required, implementation, automated)
The character \ shall not occur as a last character of a C++ comment.

Rationale

If the last character in a single-line C++ comment is \, then the comment will continue in the next line. This may lead to sections of code that are unexpectedly commented out.

Example

```
1  // $Id: A2-7-1.cpp 305382 2018-01-26 06:32:15Z michal.szczepankiewicz $
2  #include <cstdint>
3  void Fn() noexcept
4  {
5      std::int8_t idx = 0;
6      // Incrementing idx before the loop starts // Requirement X.X.X \\
7      ++idx;  // Non-compliant - ++idx was unexpectedly commented-out because of \
       character occurrence in the end of C++ comment
8
9      constexpr std::int8_t limit = 10;
10     for (; idx <= limit; ++idx)
11     {
12         // ...
13     }
14 }
```

See also

- none

Rule A2-7-2 (required, implementation, non-automated)
Sections of code shall not be “commented out”.

Rationale

Comments, using both C-style and C++ comments, should only be used to explain aspect of the source code. Code that is commented-out may become out of date, which may lead to confusion while maintaining the code.

Additionally, C-style comment markers do not support nesting, and for this purpose commenting out code is dangerous, see: M2-7-1.

Note that the code that is a part of a comment (e.g. for clarification of the usage of the function, for specifying function behavior) does not violate this rule. As it is not possible to determine if a commented block is a textual comment, a code example or a commented-out piece of code, this rule is not enforceable by static analysis tools.

Example

```
// $Id: A2-7-2.cpp 305382 2018-01-26 06:32:15Z michal.szczepankiewicz $
#include <cstdint>
void Fn1() noexcept
{
    std::int32_t i = 0;
    //    /*
    //     *  ++i; /* incrementing the variable i */
    //     */  // Non-compliant - C-style comments nesting is not supported,
    //     compilation error
    for (; i < 10; ++i)
    {
        // ...
    }
}
void Fn2() noexcept
{
    std::int32_t i = 0;
    // ++i;  // Incrementing the variable i  // Non-compliant - code should not
    // be commented-out
    for (; i < 10; ++i)
    {
        // ...
    }
}
void Fn3() noexcept
{
    std::int32_t i = 0;
    ++i;  // Incrementing the variable i using ++i syntax  // Compliant - code
          // is not commented-out, but ++i occurs in a
          // comment too
    for (; i < 10; ++i)
    {
        // ...
    }
}
```

See also

- MISRA C++ 2008 [7]: Rule 2-7-2 Sections of code shall not be “commented out” using C-style comments.

- MISRA C++ 2008 [7]: Rule 2-7-3 Sections of code should not be “commented out” using C++ comments.

Rule A2-7-3 (required, implementation, automated)
All declarations of “user-defined” types, static and non-static data members, functions and methods shall be preceded by documentation.

Rationale

Every declaration needs to provide a proper documentation.

This is compatible with the C++ standard library documentation. This forces a programmer to provide a clarification for defined types and its data members responsibilities, methods and functions usage, their inputs and outputs specification (e.g. memory management, ownership, valid boundaries), and exceptions that could be thrown.

Example

```
1  //% $Id: A2-7-3.hpp 305382 2018-01-26 06:32:15Z michal.szczepankiewicz $
2  #include <cstdint>
3
4  void F1(std::int32_t) noexcept;  // Non-compliant documentation
5
6  std::int32_t F2(std::int16_t input1,
7                  std::int32_t input2);  // Non-compliant documentation
8
9  /// @brief Function description
10 ///
11 /// @param input1  input1 parameter description
12 /// @param input2  input2 parameter description
13 /// @throw std::runtime_error  conditions to runtime_error occur
14 ///
15 /// @return return value description
16 std::int32_t F3(
17     std::int16_t input1,
18     std::int16_t input2) noexcept(false);  // Compliant documentation
19
20 /// @brief Class responsibility
21 class C  // Compliant documentation
22 {
23   public:
24     /// @brief Constructor description
25     ///
26     /// @param input1  input1 parameter description
27     /// @param input2  input2 parameter description
28     C(std::int32_t input1, float input2) : x{input1}, y{input2} {}
29
30     /// @brief Method description
31     ///
```

```
32      /// @return return value descrption
33      std::int32_t const* GetX() const noexcept { return &x; }
34
35    private:
36      /// @brief Data member descpription
37      std::int32_t x;
38      /// @brief Data member descpription
39      float y;
40  };
```

See also

- JSF December 2005 [8]: AV Rule 129: Comments in header files should describe the externally visible behavior of the functions or classes being documented.
- none

Rule A2-7-5 (required, implementation, non-automated)
Comments shall not document any actions or sources (e.g. tables, figures, paragraphs, etc.) that are outside of the file.

Rationale

Commenting only actions and sources that are inside a particular file reduce dependencies among files. Comments in a file will require changes only when content of the file reworked.

Note: This rule does not affect valid assumptions or preconditions for entities within the file.

See also

- JSF December 2005 [8]: AV Rule 128: Comments that document actions or sources (e.g. tables, figures, paragraphs, etc.) outside of the file being documented will not be allowed.

6.2.8 Header names

Rule A2-8-1 (required, architecture / design / implementation, non-automated)
A header file name should reflect the logical entity for which it provides declarations.

Rationale

Naming a header file with a name of a declared type or accordingly to a collection of free functions or forwarded headers makes include-directives and a project structure more clear and readable.

See also

- JSF December 2005 [8]: AV Rule 55: The name of a header file should reflect the logical entity for which it provides declarations.

Rule A2-8-2 (advisory, architecture / design / implementation, non-automated)
An implementation file name should reflect the logical entity for which it provides definitions.

Rationale

Naming an implementation file with a name of a declared type or accordingly to a collection of free functions makes a project structure more clear and readable.

See also

- JSF December 2005 [8]: AV Rule 56: The name of an implementation file should reflect the logical entity for which it provides definitions and have a ".cpp" extension (this name will normally be identical to the header file that provides the corresponding declarations.).

6.2.10 Identifiers

Rule M2-10-1 (required, architecture / design / implementation, automated)
Different identifiers shall be typographically unambiguous.

See MISRA C++ 2008 [7]

Rule A2-10-1 (required, architecture / design / implementation, automated)
An identifier declared in an inner scope shall not hide an identifier declared in an outer scope.

Rationale

If an identifier is declared in an inner scope and it uses the same name as an identifier that already exists in an outer scope, then the innermost declaration will "hide" the outer one. This may lead to developer confusion. The terms outer and inner scope are defined as follows:

- Identifiers that have file scope can be considered as having the outermost scope.
- Identifiers that have block scope have a more inner scope.
- Successive, nested blocks, introduce more inner scopes.

Note that declaring identifiers in different named namespaces, classes, structs or enum classes will not hide other identifiers from outer scope, because they can be accessed using fully-qualified id.

Exception

An identifier declared within a namespace using the same name as an identifier of the containing namespace does not violate the rule.

An identifier declared locally inside a lambda expression and not referring to a name of a captured variable does not violate the rule.

Example

```
1  //% $Id: A2-10-1.cpp 313834 2018-03-27 11:35:19Z michal.szczepankiewicz $
2  #include <cstdint>
3  std::int32_t sum = 0;
4  namespace
5  {
6  std::int32_t sum;  // Non-compliant, hides sum in outer scope
7  }
8  class C1
9  {
10     std::int32_t sum;  // Compliant, does not hide sum in outer scope
11 };
12 namespace n1
13 {
14     std::int32_t sum;  // Compliant, does not hide sum in outer scope
15     namespace n2
16     {
17         std::int32_t sum;  // Compliant, does not hide sum in outer scope
18     }
19 }
20
21 std::int32_t idx;
22 void F1(std::int32_t idx)
23 {
24     //Non-compliant, hides idx in outer scope
25 }
26
27 void F2()
28 {
29     std::int32_t max = 5;
30
31     for (std::int32_t idx = 0; idx < max;
32          ++idx)  // Non-compliant, hides idx in outer scope
33     {
34         for (std::int32_t idx = 0; idx < max;
35              ++idx)  // Non-compliant, hides idx in outer scope
36         {
37         }
38     }
```

```
}
void F3()
{
    std::int32_t i = 0;
    std::int32_t j = 0;
    auto lambda = [i]() {
        std::int32_t j =
            10;  // Compliant - j was not captured, so it does not hide
                 // j in outer scope
        return i + j;
    };
}
```

See also

- MISRA C++ 2008 [7]: required 2-10-2 Identifiers declared in an inner scope shall not hide an identifier declared in an outer scope.
- JSF December 2005 [8]: 4.15 AV Rule 135 Identifiers in an inner scope shall not use the same name as an identifier in an outer scope, and therefore hide that identifier.
- HIC++ v4.0 [9]: 3.1.1 Do not hide declarations.

Rule A2-10-6 (required, implementation, automated)
A class or enumeration name shall not be hidden by a variable, function or enumerator declaration in the same scope.

Rationale

C++ Language Standard [3] defines that a class or enumeration name can be hidden by an explicit declaration (of the same name) of a variable, data member, function, or enumerator in the same scope, regardless of the declaration order. Such declarations can be misleading for a developer and can lead to compilation errors.

Example

```
//% $Id: A2-10-6.cpp 313821 2018-03-27 11:16:14Z michal.szczepankiewicz $
#include <cstdint>

namespace NS1 {
class G {};
void G() {} //non-compliant, hides class G
}

namespace NS2 {
enum class H { VALUE=0, };
std::uint8_t H = 17; //non-compliant, hides
                     //scoped enum H
}
```

```
14
15 namespace NS3 {
16 class J {};
17 enum H //does not hide NS2::H, but non-compliant to A7-2-3
18 {
19     J=0, //non-compliant, hides class J
20 };
21 }
22
23 int main(void)
24 {
25     NS1::G();
26     //NS1::G a; //compilation error, NS1::G is a function
27                 //after a name lookup procedure
28     class NS1::G a{}; //accessing hidden class type name
29
30     enum NS2::H b ; //accessing scoped enum NS2::H
31     NS2::H = 7;
32
33     class NS3::J c{}; //accessing hidden class type name
34     std::uint8_t z = NS3::J;
35 }
```

See also

- ISO/IEC 14882:2014 [3]: 3.3.10.2: [basic.scope.hiding]
- MISRA C++ 2008 [7]: 2-10-6: If an identifier refers to a type, it shall not also refer to an object or a function in the same scope.
- HIC++ v4.0 [9]: 3.1.1: Do not hide declarations.

Rule A2-10-4 (required, implementation, automated)
The identifier name of a non-member object with static storage duration or static function shall not be reused within a namespace.

Rationale

No identifier with static storage duration should be re-used in the same namespace across any source files in the project.

This may lead to the developer or development tool confusing the identifier with another one.

Example

```
1 //% $Id: A2-10-4.cpp 305382 2018-01-26 06:32:15Z michal.szczepankiewicz $
2 #include <cstdint>
3 // f1.cpp
4 namespace ns1
5 {
```

```
static std::int32_t globalvariable = 0;
}

// f2.cpp
namespace ns1
{
// static std::int32_t globalvariable = 0; // Non-compliant - identifier reused
// in ns1 namespace in f1.cpp
}
namespace ns2
{
static std::int32_t globalvariable =
    0;  // Compliant - identifier reused, but in another namespace
}

// f3.cpp
static std::int32_t globalvariable =
    0;  // Compliant - identifier reused, but in another namespace
```

See also

- MISRA C++ 2008 [7]: advisory 2-10-5 The identifier name of a non-member object or function with static storage duration should not be reused.

Rule A2-10-5 (advisory, design / implementation, automated)
An identifier name of a function with static storage duration or a non-member object with external or internal linkage should not be reused.

Rationale

Regardless of scope, no identifier with static storage duration should be re-used across any source files in the project. This includes objects or functions with external linkage and any objects or functions with static storage class specifier. While the compiler can understand this, the possibility exists for the developer or development tool to incorrectly associate unrelated variables with the same name.

Note: This rule does not apply to objects without linkage, e.g. function local static objects.

Example

```
//% $Id: A2-10-5.cpp 305382 2018-01-26 06:32:15Z michal.szczepankiewicz $
#include <cstdint>
// f1.cpp
namespace n_s1
{
static std::int32_t globalvariable = 0;
}
static std::int32_t filevariable = 5;  // Compliant - identifier not reused
static void Globalfunction();
```

```
10
11 // f2.cpp
12 namespace n_s1
13 {
14 // static std::int32_t globalvariable = 0; // Non-compliant - identifier reused
15 static std::int16_t modulevariable = 10;  // Compliant - identifier not reused
16 }
17 namespace n_s2
18 {
19 static std::int16_t modulevariable2 = 20;
20 }
21 static void Globalfunction();              // Non-compliant - identifier reused
22 static std::int16_t modulevariable2 = 15;  // Non-compliant - identifier reused
```

See also

- MISRA C++ 2008 [7]: advisory 2-10-5 The identifier name of a non-member object or function with static storage duration should not be reused.

6.2.11 Keywords

Rule A2-11-1 (required, design / implementation, automated)
Volatile keyword shall not be used.

Rationale

The volatile keyword disables compiler optimizations for a particular variable or object's value in case those values may change in ways not specified by the language (e.g. object representing a hardware register). It is error prone and often misused by developers, as they expect this is equal to variable or object's value being atomic.

Note: The main intention of this rule is to eliminate incorrect usages of volatile keyword and force developers to precisely document each usage of volatile keyword.

See also

- JSF December 2005 [8]: AV Rule 205: The volatile keyword shall not be used unless directly interfacing with hardware.
- HIC++ v4.0 [9]: 18.2.3: Do not share volatile data between threads.
- C++ Core Guidelines [11]: CP.8: Don't try to use volatile for synchronization.
- C++ Core Guidelines [11]: CP.200: Use volatile only to talk to non-C++ memory.

6.2.13 Literals

Rule A2-13-1 (required, architecture / design / implementation, automated) Only those escape sequences that are defined in ISO/IEC 14882:2014 shall be used.

Rationale

The use of an undefined escape sequence leads to undefined behavior. The defined escape sequences (ISO/IEC 14882:2014) are: \', \", \?, \\, \a, \b, \f, \n, \r, \t, \v, \<Octal Number>, \x<Hexadecimal Number>.

Note: Universal-character-names (\u hex-quad and \U hex-quad hex-quad) are also allowed in character and string literals (although they look similar to escape sequences, they are handled in a different way by the C++ language, see A2-13-6).

Example

```
1 //% $Id: A2-13-1.cpp 305629 2018-01-29 13:29:25Z piotr.serwa $
2 #include <string>
3 void F()
4 {
5     const std::string a = "\k";          // Non-compliant
6     const std::string b = "\n";          // Compliant
7     const std::string c = "\U0001f34c";  // Compliant
8 }
```

See also

- MISRA C++ 2008 [7]: required 2-13-1 Only those escape sequences that are defined in ISO/IEC14882:2003 shall be used.

Rule A2-13-6 (required, architecture / design / implementation, automated) Universal character names shall be used only inside character or string literals.

Rationale

Using universal-character-names to define a language identifier can be confusing for a developer and may be troublesome to use this identifier in the source code.

Example

```
1 //% $Id: A2-13-6.cpp 307578 2018-02-14 14:46:20Z michal.szczepankiewicz $
2 #include <string>
3 void F()
4 {
5     const std::string c = "\U0001f34c";  // Compliant
6 }
7
8 //non-compliant
9 void \U0001f615()
```

```
{
  //
}
```

Rule A2-13-5 (advisory, implementation, automated)
Hexadecimal constants should be upper case.

Rationale

Using upper case literals for hexadecimal constants makes the source code consistent in this matter and removes a potential developer confusion.

Example

```
//% $Id: A2-13-5.cpp 305629 2018-01-29 13:29:25Z piotr.serwa $

#include <cstdint>

int main(void)
{
    std::int16_t a = 0x0f0f; //non-compliant
    std::int16_t b = 0x0f0F; //non-compliant
    std::int16_t c = 0x0F0F; //compliant

    return 0;
}
```

See also

- JSF December 2005 [8]: AV Rule 150: Hexadecimal constants will be represented using all uppercase letters.

Rule M2-13-2 (required, architecture / design / implementation, automated)
Octal constants (other than zero) and octal escape sequences (other than "\0") shall not be used.

See MISRA C++ 2008 [7]

Rule M2-13-3 (required, architecture / design / implementation, automated)
A "U" suffix shall be applied to all octal or hexadecimal integer literals of unsigned type.

See MISRA C++ 2008 [7]

Rule M2-13-4 (required, architecture / design / implementation, automated)
Literal suffixes shall be upper case.

See MISRA C++ 2008 [7]

Rule A2-13-2 (required, implementation, automated)
String literals with different encoding prefixes shall not be concatenated.

Rationale

Concatenation of wide and narrow string literals leads to undefined behavior.

"In translation phase 6 (2.2), adjacent string-literals are concatenated. If both string-literals have the same encoding-prefix, the resulting concatenated string literal has that encoding-prefix. If one string-literal has no encoding-prefix, it is treated as a string-literal of the same encoding-prefix as the other operand. If a UTF-8 string literal token is adjacent to a wide string literal token, the program is ill-formed. Any other concatenations are conditionally-supported with implementation-defined behavior. [Note: This concatenation is an interpretation, not a conversion. Because the interpretation happens in translation phase 6 (after each character from a literal has been translated into a value from the appropriate character set), a string-literal's initial rawness has no effect on the interpretation or well-formedness of the concatenation. -end note]" [C++14 Language Standard] [3]

Example

```
1  //% $Id: A2-13-2.cpp 305629 2018-01-29 13:29:25Z piotr.serwa $
2
3  char16_t nArray[] =
4      u"Hello"
5      u"World";  // Compliant, "u" stands for char16_t type
6
7  char32_t nArray2[] =
8      U"Hello"
9      U"World";  // Compliant, "U" stands for char32_t type
10
11 wchar_t wArray[] =
12     L"Hello"
13     L"World";  // Compliant, "L" stands for wchar_t type - violates A2-13-3
14                // rule.
15
16 wchar_t mixed1[] =
17     "Hello"
18     L"World";  // Compliant
19
20 char32_t mixed2[] =
21     "Hello"
22     U"World";  // Compliant
```

```
23
24 char16_t mixed3[] =
25     "Hello"
26     u"World";  // Compliant
27
28 // wchar_t mixed1[] = u"Hello" L"World";  // Non-compliant - compilation error
29
30 // char32_t mixed2[] = u"Hello" U"World";  // Non-compliant - compilation error
```

See also

- MISRA C++ 2008 [7]: required 2-13-5 Narrow and wide string literals shall not be concatenated.
- HIC++ v4.0 [9]: 2.5.1 Do not concatenate strings with different encoding prefixes

Rule A2-13-3 (required, architecture / design / implementation, automated)
Type wchar_t shall not be used.

Rationale

Width of wchar_t type is implementation-defined.

Types char16_t and char32_t should be used instead.

Example

```
1 //% $Id: A2-13-3.cpp 305629 2018-01-29 13:29:25Z piotr.serwa $
2 char16_t string1[] = u"ABC";  // Compliant
3 char32_t string2[] = U"DEF";  // Compliant
4 wchar_t string3[] = L"GHI";   // Non-compliant
```

See also

- none

Rule A2-13-4 (required, architecture / design / implementation, automated)
String literals shall not be assigned to non-constant pointers.

Rationale

Since C++0x, there was a change in subclause 2.13.5 for string literals. To prevent from calling an inappropriate function that might modify its argument, the type of a string literal was changed from “array of char” to “array of const char”.

Such a usage is deprecated by the Standard and reported by a compiler as a warning. This rule is deliberately redundant, in case rules A1-1-1 and A1-4-3 are disabled in a project.

Example

```
//% $Id: A2-13-4.cpp 307578 2018-02-14 14:46:20Z michal.szczepankiewicz $

int main(void)
{
    char* nc1 = "AUTOSAR";        //non-compliant
    char nc2[] = "AUTOSAR";       //compliant with A2-13-4, non-compliant with A18-1-1
    char nc3[8] = "AUTOSAR";      //compliant with A2-13-4, non-compliant with A18-1-1
    nc1[3] = 'a';   // undefined behaviour
    const char* c1 = "AUTOSAR";        //compliant
    const char c2[] = "AUTOSAR";       //compliant with A2-13-4, non-compliant with A18-1-1
    const char c3[8] = "AUTOSAR";      //compliant with A2-13-4, non-compliant with A18-1-1
    //c1[3] = 'a';  //compilation error

    return 0;
}
```

See also

- JSF December 2005 [8]: AV Rule 151.1: A string literal shall not be modified.

6.3 Basic concepts

6.3.1 Declarations and definitions

Rule A3-1-1 (required, architecture / design / implementation, automated)
It shall be possible to include any header file in multiple translation units without violating the One Definition Rule.

Rationale

A header file is a file that holds declarations used in more than one translation unit and acts as an interface between separately compiled parts of a program. A header file often contains classes, object declarations, enums, functions, inline functions, templates, typedefs, type aliases and macros.

In particular, a header file is not supposed to contain or produce definitions of global objects or functions that occupy storage, especially objects that are not declared "extern" or definitions of functions that are not declared "inline".

Example

```
//% $Id: A3-1-1.hpp 289436 2017-10-04 10:45:23Z michal.szczepankiewicz $
#include <cstdint>
void F1();          // Compliant
```

```
4  extern void F2();  // Compliant
5  void F3()
6  {
7  }  // Non-compliant
8  static inline void F4()
9  {
10 }  // Compliant
11 template <typename T>
12 void F5(T)
13 {
14 }  // Compliant
15 std::int32_t a;                          // Non-compliant
16 extern std::int32_t b;                   // Compliant
17 constexpr static std::int32_t c = 10;   // Compliant
18 namespace ns
19 {
20 constexpr static std::int32_t d = 100;  // Compliant
21 const static std::int32_t e = 50;       // Compliant
22 static std::int32_t f;                  // Non-compliant
23 static void F6() noexcept;              // Non-compliant
24 }
```

See also

- MISRA C++ 2008 [7]: Rule 3-1-1 It shall be possible to include any header file in multiple translation units without violating the One Definition Rule.

Rule A3-1-2 (required, architecture / design / implementation, automated) Header files, that are defined locally in the project, shall have a file name extension of one of: ".h", ".hpp" or ".hxx".

Rationale

This is consistent with developer expectations to provide header files with one of the standard file name extensions.

Example

```
1  //% $Id: A3-1-2.cpp 266557 2017-02-07 13:08:19Z piotr.tanski $
2  //#include <h3.h>    // Compliant
3  //#include <h1.hpp>  // Compliant
4  //#include <h2.hxx>  // Compliant
5  //#include <h4.cpp>  // Non-compliant
6  //#include <h5.c>    // Non-compliant
7  //#include <h6.hdr>  // Non-compliant
8  //#include <h7.inc>  // Non-compliant
```

See also

- JSF December 2005 [8]: 4.9.2 AV Rule 53 Header files will always have a file name extension of ".h".

Rule A3-1-3 (advisory, architecture / design / implementation, automated)
Implementation files, that are defined locally in the project, should have a file name extension of ".cpp".

Rationale

This is consistent with developer expectations to provide C++ implementation files with the standard file name extension.

Note that compilers support various file name extensions for C++ implementation files.

See also

- JSF December 2005 [8]: 4.9.2 AV Rule 54 Implementation files will always have a file name extension of ".cpp".

Rule M3-1-2 (required, implementation, automated)
Functions shall not be declared at block scope.

See MISRA C++ 2008 [7]

Rule A3-1-4 (required, design / implementation, automated)
When an array with external linkage is declared, its size shall be stated explicitly.

Rationale

Although it is possible to declare an array of incomplete type and access its elements, it is safer to do so when the size of the array can be explicitly determined.

Example

```
1 //% $Id: A3-1-4.hpp 271687 2017-03-23 08:57:35Z piotr.tanski $
2 #include <cstdint>
3 extern std::int32_t array1[];    // Non-compliant
4 extern std::int32_t array2[42];  // Compliant
```

See also

- MISRA C++ 2008 [7]: Rule 3-1-3 When an array is declared, its size shall either be stated explicitly or defined implicitly by initialization.

Rule A3-1-5 (required, design, partially-automated)
A function definition shall only be placed in a class definition if (1) the function is intended to be inlined (2) it is a member function template (3) it is a member function of a class template.

Rationale

Merging the implementation into the declaration instructs a compiler to inline the method which may save both time and space for short functions. For templates, it allows to reduce repetitions of template syntax elements (e.g. parameter list), which makes code less difficult to read and maintain.

Example

```
1  //% $Id: A3-1-5.cpp 305629 2018-01-29 13:29:25Z piotr.serwa $
2  #include <cstdint>
3  #include <iostream>
4
5  class A
6  {
7  public:
8      //compliant with (2)
9      template <typename T>
10     void Foo(T&& t)
11     {
12         std::cout << __PRETTY_FUNCTION__ << " defined inside with param: " << t
       << std::endl;
13     }
14
15     //non-compliant with (2)
16     template <typename T>
17     void Bar(T&& t);
18
19     //compliant with (1)
20     std::uint32_t GetVal() const noexcept
21     {
22         return val;
23     }
24
25     //non-compliant with (1)
26     std::uint32_t GetVal2() const noexcept;
27
28 private:
29     std::uint32_t val = 5;
30 };
31
32
33 template <typename T>
34 void A::Bar(T&& t)
35 {
36     std::cout << __PRETTY_FUNCTION__ << " defined outside with param: " << t <<
       std::endl;
37 }
38
39 std::uint32_t A::GetVal2() const noexcept
40 {
```

```
41      return val;
42  }
43
44  template <typename T>
45  class B
46  {
47  public:
48      B(const T& x) : t(x) {}
49
50      //compliant with (3)
51      void display() const noexcept
52      {
53          std::cout << t << std::endl;
54      }
55
56      //non-compliant with (3)
57      void display2() const noexcept;
58
59  private:
60      T t;
61  };
62
63  template <typename T>
64  void B<T>::display2() const noexcept
65  {
66      std::cout << t << std::endl;
67  }
68
69  int main(void)
70  {
71      std::uint32_t tmp = 5;
72      A a;
73      a.Foo(3.14f);
74      a.Bar(5);
75
76      std::cout << a.GetVal() << std::endl;
77
78      B<std::int32_t> b(7);
79      b.display();
80
81      return 0;
82  }
```

See also

- JSF December 2005 [8]: AV Rule 109: A function definition should not be placed in a class specification unless the function is intended to be inlined.

Rule A3-1-6 (advisory, design, automated)
Trivial accessor and mutator functions should be inlined.

Rationale

Inlining trivial accessors and mutators saves time and space, as it reduces multiple syntax elements that has to be repeated.

Example

```
//% $Id: A3-1-6.cpp 305629 2018-01-29 13:29:25Z piotr.serwa $
#include <cstdint>

class A
{
public:
    A(std::int32_t l) noexcept : limit{l} {}
    //compliant
    std::int32_t Limit() const noexcept { return limit; }
    //compliant
    void SetLimit(std::int32_t l) { limit = l; }

    //non-compliant
    //std::int32_t Limit() const noexcept
    //{
        //open file, read data, close file
        //return value
    //}
    //non-compliant
    //void SetLimit(std::int32_t l)
    //{
        //open file, write data, close file
    //}

private:
    std::int32_t limit;
};
```

See also

- JSF December 2005 [8]: AV Rule 122: Trivial accessor and mutator functions should be inlined.

6.3.2 One Definition Rule

Rule M3-2-1 (required, implementation, automated)
All declarations of an object or function shall have compatible types.

See MISRA C++ 2008 [7]

Rule M3-2-2 (required, implementation, automated)
The One Definition Rule shall not be violated.

See MISRA C++ 2008 [7]

Rule M3-2-3 (required, implementation, automated)
A type, object or function that is used in multiple translation units shall be declared in one and only one file.

See MISRA C++ 2008 [7]

Rule M3-2-4 (required, implementation, automated)
An identifier with external linkage shall have exactly one definition.

See MISRA C++ 2008 [7]

6.3.3 Scope

Rule A3-3-1 (required, implementation, automated)
Objects or functions with external linkage (including members of named namespaces) shall be declared in a header file.

Rationale

Placing the declarations of objects and functions with external linkage in a header file means that they are intended to be accessible from other translation units.

If external linkage is not needed, then the object or function is supposed to be either declared in an unnamed namespace or declared static in the implementation file. This reduces the visibility of objects and functions, which allows to reach a higher encapsulation and isolation.

Note that members of named namespace are by default external linkage objects.

Exception

This rule does not apply to main, or to members of unnamed namespaces.

Example

```
1 //% $Id: A3-3-1.hpp 289436 2017-10-04 10:45:23Z michal.szczepankiewicz $
2 #include <cstdint>
3 extern std::int32_t a1;
4 extern void F4();
```

```
namespace n
{
void F2();
std::int32_t a5;  // Compliant, external linkage
}
```

```
//% $Id: A3-3-1.cpp 289436 2017-10-04 10:45:23Z michal.szczepankiewicz $
#include "A3-3-1.hpp"
std::int32_t a1 = 0;          // Compliant, external linkage
std::int32_t a2 = 0;          // Non-compliant, static keyword not used
static std::int32_t a3 = 0;   // Compliant, internal linkage
namespace
{
std::int32_t a4 = 0;  // Compliant by exception
void F1()             // Compliant by exception
{
}
}
namespace n
{
void F2()  // Compliant, external linkage
{
}
std::int32_t a6 = 0;  // Non-compliant, external linkage
}
extern std::int32_t a7;  // Non-compliant, extern object declared in .cpp file
static void F3()         // Compliant, static keyword used
{
}
void F4()  // Compliant, external linkage
{
    a1 = 1;
    a2 = 1;
    a3 = 1;
    a4 = 1;
    n::a5 = 1;
    n::a6 = 1;
    a7 = 1;
}
void F5()  // Non-compliant, static keyword not used
{
    a1 = 2;
    a2 = 2;
    a3 = 2;
    a4 = 2;
    n::a5 = 2;
    n::a6 = 2;
    a7 = 2;
}
int main(int, char**)  // Compliant by exception
{
    F1();
```

```
47      n::F2();
48      F3();
49      F4();
50      F5();
51  }
```

See also

- MISRA C++ 2008 [7]: Rule 3-3-1 Objects or functions with external linkage shall be declared in a header file.

Rule A3-3-2 (required, implementation, automated)
Static and thread-local objects shall be constant-initialized.

Rationale

In general, using non-const global and static variables obscures the true dependencies of an API, since they can be accessed from any place of the source code. It therefore makes the code more difficult to maintain, less readable, and significantly less testable.

A particular problem is that the order in which constructors and initializers for static variables are called is only partially specified by the C++ Language Standard and can even change from build to build. This can cause issues that are difficult to find or debug.

The compiler performs constant-initialization, if

- the object is initialized by a `constexpr` constructor with only constant expression as arguments; or
- the object is not initialized by a constructor call, but is value-initialized (*`T object{};`*); or
- the object is not initialized by a constructor call, but is initialized by an initializer consisting only of constant expressions.

Constant initialization is guaranteed to occur before any other initialization of static or thread-local objects and may happen at compile time. Thus it is guaranteed that problematic dependencies between the initializers of constant-initialized static or thread-local objects cannot occur.

Note that declaring a static variable as `constexpr` (`static` is implied in this case, but may be added to the declaration), enforces constant initialization by the compiler.

Note that the rule applies to:

- global variables (i.e. extern)
- static variables
- static class member variables
- static function-scope variables

Example

```
// $Id: A3-3-2.cpp 305690 2018-01-29 14:35:00Z jan.babst $
#include <cstdint>
#include <limits>
#include <string>
class A
{
  public:
    static std::uint8_t instanceId;
    static float const pi;
    static std::string const separator;

    A() {}
    // Implementation...
};
std::uint8_t A::instanceId = 0;     // Compliant - constant initialization
float const A::pi = 3.14159265359;  // Compliant - constant initialization
std::string const A::separator =
    "==========";  // Non-compliant - string c'tor is not constexpr

class C
{
  public:
    constexpr C() = default;
};

namespace
{
constexpr std::int32_t maxInt32 =
    std::numeric_limits<std::int32_t>::max();  // Compliant - constexpr variable

A instance{};     // Compliant - constant (value) initialization
constexpr C c{};  // Compliant - constexpr c'tor call
}  // namespace

void Fn() noexcept
{
    static A a{};  // Non-compliant - A's default c'tor is not constexpr
    static std::int32_t counter{0};     // Compliant
    static std::string border(5, '*');  // Non-compliant - not a constexpr c'tor
}

class D
{
  public:
    D() = default;
    D(D const&) = default;
    D(D&&) = default;
    D& operator=(D const&) = default;
    D& operator=(D&&) = default;
    ~D() = default;
```

```
51
52    private:
53      static D* instance;
54  };
55  D* D::instance = nullptr;  // Compliant - initialization by constant expression
```

See also

- cppreference.com [16]: Constant initialization.
- HIC++ v4.0 [9]: 3.3.1: Do not use variables with static storage duration.
- JSF December 2005 [8]: AV Rule 214: Assuming that non-local static objects, in separate translation units, are initialized in a special order shall not be done.
- SEI CERT C++ Coding Standard [10]: DCL56-CPP: Avoid cycles during initialization of static objects.
- C++ Core Guidelines [11]: I.22: Avoid complex initialization of global objects.
- Google C++ Style Guide [12]: Static and Global Variables.

Rule M3-3-2 (required, implementation, automated)
If a function has internal linkage then all re-declarations shall include the static storage class specifier.

See MISRA C++ 2008 [7]

Note: Static storage duration class specifier is redundant and does not need to be specified if a function is placed in an unnamed namespace.

6.3.4 Name lookup

Rule M3-4-1 (required, implementation, automated)
An identifier declared to be an object or type shall be defined in a block that minimizes its visibility.

See MISRA C++ 2008 [7]

See also

- C++ Core Guidelines [11]: ES.21: Don't introduce a variable (or constant) before you need to use it.

6.3.8 Object lifetime

Rule A3-8-1 (required, implementation, not automated)
An object shall not be accessed outside of its lifetime.

Rationale

Accessing an object outside of its lifetime, i.e. before its initialization or constructor has completed, or after its non-trivial destructor has finished, is well defined only for a very restricted number of cases, as laid out by the language standard. Outside of these cases it leads to undefined behavior.

Note: The AUTOSAR C++14 guidelines contain several other rules which are special cases of this rule (see references below). This rule was added to provide generic coverage for all cases not contained in these specialized rules. This also makes it easier to provide tracing from other standards with a similar generic rule.

Note: The examples given below are not intended to represent a complete list of situations where violations of this rule can occur.

Example

```
1  //% $Id: A3-8-1.cpp 305786 2018-01-30 08:58:33Z michal.szczepankiewicz $
2
3  //
4  // 1. Pointer to virtual base is passed as function argument after lifetime of
5  // object has ended.
6  //
7
8  class B
9  {
10 };
11
12 class C1 : public virtual B  // violates M10-1-1
13 {
14 };
15
16 class C2 : public virtual B  // violates M10-1-1
17 {
18 };
19
20 class D : public C1, public C2
21 {
22 };
23
24 void f(B const* b){};
25
26 void example1()
27 {
28     D* d = new D();  // lifetime of d starts (violates A18-5-2)
29     // Use d
30     delete d;  // lifetime of d ends (violates A18-5-2)
31
```

```
32      f(d);  // Non-compliant - Undefined behavior, even if argument is not used
33             // by f().
34  }
35
36  //
37  // 2. Accessing an initializer_list after lifetime of initializing array has
38  // ended.
39  //
40  class E
41  {
42      std::initializer_list<int> lst;
43
44    public:
45      // Conceptually, this works as if a temporary array {1, 2, 3} was created
46      // and a reference to this array was passed to the initializer_list. The
47      // lifetime of the temporary array ends when the constructor finishes.
48      E() : lst{1, 2, 3} {}
49
50      int first() const { return *lst.begin(); }
51  };
52
53  void example2()
54  {
55      E e;
56      std::out << e.first() << "\n";  // Non-compliant
57  }
58
59  //
60  // 3. Exiting main while running tasks depend on static objects
61  //
62  void initialize_task()
63  {
64      // start some task (separate thread) which depends on some static object.
65      // ...
66  }
67
68  int main()
69  {
70      // static constructors are called
71
72      initialize_task();
73  }  // main ends, static destructors are called
74
75  // Non-compliant
76  // Task begins to run and accesses destroyed static object.
77
78  //
79  // 4. Storage reuse without explicit destructor call
80  //
81  void example4()
82  {
```

```
83      std::string str;
84      new (&a) std::vector<int>{};  // Non-compliant: storage of str reused without
85                                    // calling its non-trivial destructor.
86  }   // Non-compliant: Destructor of str is implicitly called at scope exit, but
87      // storage contains object of different type.
```

See also

- ISO/IEC 14882:2014 [3]: 3.8: [basic.life]
- JSF December 2005 [8]: AV Rule 70.1: An object shall not be improperly used before its lifetime begins or after its lifetime ends.
- SEI CERT C++ Coding Standard [10]: EXP54-CPP: Do not access an object outside of its lifetime.
- A5-1-4 in section 6.5.1
- M7-5-1 in section 6.7.5
- M7-5-2 in section 6.7.5
- A7-5-1 in section 6.7.5
- M12-1-1 in section 6.12.1

6.3.9 Types

Rule M3-9-1 (required, implementation, automated)
The types used for an object, a function return type, or a function parameter shall be token-for-token identical in all declarations and re-declarations.

See MISRA C++ 2008 [7]

Rule A3-9-1 (required, implementation, automated)
Fixed width integer types from <cstdint>, indicating the size and signedness, shall be used in place of the basic numerical types.

Rationale

The basic numerical types of char, int, short, long are not supposed to be used, specific-length types from <cstdint> header need be used instead.

Fixed width integer types are:

- std::int8_t
- std::int16_t

- std::int32_t
- std::int64_t
- std::uint8_t
- std::uint16_t
- std::uint32_t
- std::uint64_t

Exception

The wchar_t does not need a typedef as it always maps to a type that supports wide characters.

Example

```
1  //% $Id: A3-9-1.cpp 289436 2017-10-04 10:45:23Z michal.szczepankiewicz $
2  #include <cstdint>
3  void F()
4  {
5      std::int32_t i1 = 5;     // Compliant
6      int i2 = 10;             // Non-compliant
7      std::int64_t i3 = 250;   // Compliant
8      long int i4 = 50;        // Non-compliant
9      std::int8_t i5 = 16;     // Compliant
10     char i6 = 23;            // Non-compliant
11 }
```

See also

- MISRA C++ 2008 [7]: Rule 3-9-2 typedefs that indicate size and signedness should be used in place of the basic numerical types.

Rule M3-9-3 (required, implementation, automated)
The underlying bit representations of floating-point values shall not be used.

See MISRA C++ 2008 [7]

6.4 Standard conversions

6.4.5 Integral promotions

Rule M4-5-1 (required, implementation, automated)
Expressions with type bool shall not be used as operands to built-in operators other than the assignment operator =, the logical operators

`&&`, `||`, `!`, the equality operators $==$ and $!=$, the unary `&` operator, and the conditional operator.

See MISRA C++ 2008 [7]

Rule A4-5-1 (required, implementation, automated)
Expressions with type enum or enum class shall not be used as operands to built-in and overloaded operators other than the subscript operator [ ], the assignment operator $=$, the equality operators $==$ and $!=$, the unary & operator, and the relational operators $<$, $<=$, $>$, $>=$.

Rationale

Enumerations, i.e. enums or enum classes, have implementation-defined representations and they are not supposed to be used in arithmetic contexts.

Note that only enums can be implicitly used as operands to other built-in operators, like operators $+$, $-$, $*$, etc. Enum class needs to provide definitions of mentioned operators in order to be used as operand.

Exception

It is allowed to use the enumeration as operand to all built-in and overloaded operators if the enumeration satisfies the “BitmaskType” concept [16].

Example

```
1  // $Id: A4-5-1.cpp 289436 2017-10-04 10:45:23Z michal.szczepankiewicz $
2  #include <cstdint>
3  enum Colour : std::uint8_t
4  {
5      Red,
6      Green,
7      Blue,
8      ColoursCount
9  };
10 void F1() noexcept(false)
11 {
12     Colour colour = Red;
13     if (colour == Green)  // Compliant
14     {
15     }
16
17     if (colour == (Red + Blue))  // Non-compliant
18     {
19     }
20
21     if (colour < ColoursCount)  // Compliant
22     {
23     }
```

```
}
enum class Car : std::uint8_t
{
    Model1,
    Model2,
    Model3,
    ModelsCount
};
void F2() noexcept(false)
{
    Car car = Car::Model1;
    if (car != Car::Model2)  // Compliant
    {
    }

    if (car == Car::Model3)  // Compliant
    {
    }

    // if (car == (Car::Model1 + Car::Model2))  // Non-compliant -
    // operator+ not provided for Car enum class, compilation error
    //{
    //}
    if (car < Car::ModelsCount)  // Compliant
    {
    }
}
Car operator+(Car lhs, Car rhs)
{
    return Car::Model3;
}
void F3() noexcept(false)
{
    Car car = Car::Model3;
    if (car == (Car::Model1 + Car::Model2))  // Non-compliant - overloaded
                                             // operator+ provided, no
                                             // compilation error
    {
    }
}
enum Team : std::uint8_t
{
    TeamMember1 = 0,
    TeamMember2 = 1,
    TeamMember3 = 2,
    TeamMember4 = 3,
    TeamMembersStart = TeamMember1,
    TeamMembersEnd = TeamMember2,
    TeamMembersCount = 4
};
void F4(const char* teamMember)
```

```
{
    // Implementation
}
void F5()
{
    const char* team[TeamMembersCount];  // Compliant
    // ...
    F4(team[TeamMember2]);  // Compliant
}
```

See also

- MISRA C++ 2008 [7]: Rule 4-5-2 Expressions with type enum shall not be used as operands to built-in operators other than the subscript operator [ ], the assignment operator =, the equality operators == and !=, the unary & operator, and the relational operators <, <=, >, >=.

Rule M4-5-3 (required, implementation, automated)
Expressions with type (plain) char and wchar_t shall not be used as operands to built-in operators other than the assignment operator =, the equality operators == and !=, and the unary & operator.

See MISRA C++ 2008 [7]

6.4.7 Integral conversion

Rule A4-7-1 (required, implementation, automated)
An integer expression shall not lead to data loss.

Rationale

Implicit conversions, casts and arithmetic expressions may lead to data loss, e.g. overflows, underflows or wrap-around.

Integral expressions need to be performed on proper integral types that ensure that the data loss will not occur or appropriate guards should be used to statically detect or counteract such a data loss.

Example

```
// $Id: A4-7-1.cpp 289436 2017-10-04 10:45:23Z michal.szczepankiewicz $
#include <cstdint>
#include <stdexcept>
std::int8_t Fn1(std::int8_t x, std::int8_t y) noexcept
{
    return (x + y);  // Non-compliant - may lead to overflow
```

```
}
std::int8_t Fn2(std::int8_t x, std::int8_t y)
{
    if (x > 100 || y > 100)  // Range check
    {
        throw std::logic_error("Preconditions check error");
    }
    return (x + y);  // Compliant - ranges of x and y checked before the
                     // arithmetic operation
}
std::int16_t Fn3(std::int8_t x, std::int8_t y) noexcept
{
    return (static_cast<std::int16_t>(x) + y);  // Compliant - std::int16_t type
                                                // is enough for this arithmetic
                                                // operation
}
std::uint8_t Fn4(std::uint8_t x, std::uint8_t y) noexcept
{
    return (x * y);  // Non-compliant - may lead to wrap-around
}
std::int8_t Fn5(std::int16_t x)
{
    return static_cast<std::int8_t>(x);  // Non-compliant - data loss
}
std::int8_t Fn6(std::int16_t x)
{
    return x;  // Non-compliant - data loss by implicit conversion
}
void F()
{
    std::int8_t x1 =
        Fn1(5, 10);  // Compliant - overflow will not occur for these values
    std::int8_t x2 = Fn1(250, 250);  // Non-compliant - Overflow occurs
    try
    {
        std::int8_t x3 =
            Fn2(250, 250);  // Compliant - No overflow, range checks
                            // inside fn2() function
    }
    catch (std::logic_error&)
    {
        // Handle an error
    }
    std::int16_t x4 = Fn3(250, 250);  // Compliant - No overflow, arithmetic
                                      // operation underlying type is wider than
                                      // std::int8_t
    std::uint8_t x5 = Fn4(50, 10);    // Non-compliant - Wrap-around occurs
    std::int8_t x6 = Fn5(100);        // Compliant - data loss will not occur
    std::int8_t x7 = Fn5(300);        // Non-compliant - Data loss occurs
    std::int8_t x8 = Fn6(300);        // Non-compliant - Data loss occurs

```

```
58      std::int8_t x9 = 150;
59      std::int16_t x10 = static_cast<std::int16_t>(x9 + x9);  // Non-compliant
60      x10 = x9 + x9;                                          // Non-compliant
61      x10 = static_cast<std::int16_t>(x9) + x9;               // Compliant
62
63      std::int8_t x11 = x9 << 5;  // Non-compliant
64
65      std::int8_t x12 = 127;
66      ++x12;  // Non-compliant
67
68      std::uint8_t x13 = 255;
69      ++x13;  // Non-compliant
70  }
```

See also

- MISRA C++ 2008 [7]: Rule 5-0-6 An implicit integral or floating-point conversion shall not reduce the size of the underlying type.
- MISRA C++ 2008 [7]: Rule 5-0-8 An explicit integral or floating-point conversion shall not increase the size of the underlying type of a cvalue expression.
- HIC++ v4.0 [9]: 4.2.2 Ensure that data loss does not demonstrably occur in an integral expression.
- JSF December 2005 [8]: AV Rule 212: Underflow or overflow functioning shall not be depended on in any special way.
- C++ Core Guidelines [11]: ES.46: Avoid lossy (narrowing, truncating) arithmetic conversions.

6.4.10 Pointer conversions

Rule M4-10-1 (required, implementation, automated)
NULL shall not be used as an integer value.

See MISRA C++ 2008 [7]

Rule A4-10-1 (required, architecture / design / implementation, automated)
Only nullptr literal shall be used as the null-pointer-constant.

Rationale

In C++, the literal NULL is both the null-pointer-constant and an integer type. To meet developer expectations, only nullptr pointer literal shall be used as the null-pointer-constant.

Note that, nullptr pointer literal allows parameters forwarding via a template function.

Example

```
//% $Id: A4-10-1.cpp 298086 2017-11-24 11:13:27Z michal.szczepankiewicz $
#include <cstddef>
#include <cstdint>

void F1(std::int32_t);
void F2(std::int32_t*);
void F3()
{
    F1(0);     // Compliant
    F1(NULL);  // Non-compliant - NULL used as an integer,
    // compilable
    // f1(nullptr); // Non-compliant - nullptr used as an integer
    // compilation error
    F2(0);         // Non-compliant - 0 used as the null pointer constant
    F2(NULL);      // Non-compliant - NULL used as the null pointer constant
    F2(nullptr);  // Compliant
}
void F4(std::int32_t*);
template <class F, class A>
void F5(F f, A a)
{
    F4(a);
}
void F6()
{
    // f5(f4, NULL);  // Non-compliant - function f4(std::int32_t) not declared
    F5(F4, nullptr);  // Compliant
}
```

See also

- HIC++ v4.0 [9]: 2.5.3 Use nullptr for the null pointer constant

Rule M4-10-2 (required, implementation, automated)
Literal zero (0) shall not be used as the null-pointer-constant.

See MISRA C++ 2008 [7]

6.5 Expressions

6.5.0 General

Rule A5-0-1 (required, implementation, automated)
The value of an expression shall be the same under any order of evaluation that the standard permits.

Rationale

Apart from a few operators (notably `&&`, `||`, `?:` and `,`) the order in which sub-expressions are evaluated is unspecified and can vary. This means that no reliance can be placed on the order of evaluation of sub-expressions and, in particular, no reliance can be placed on the order in which side effects occur. Those points in the evaluation of an expression at which all previous side effects can be guaranteed to have taken place are called "sequencing". Sequencing and side effects are described in Section 1.9(7) of ISO/IEC 14882:2014 [3].

Note that the "order of evaluation" problem is not solved by the use of parentheses, as this is not a precedence issue.

Example

```
1  // $Id: A5-0-1.cpp 289436 2017-10-04 10:45:23Z michal.szczepankiewicz $
2  #include <cstdint>
3  #include <stack>
4  // The following notes give some guidance on how dependence on order of
5  // evaluation may occur, and therefore may assist in adopting the rule.
6
7  // 1) Increment or decrement operators
8  // As an example of what can go wrong, consider
9  void F1(std::uint8_t (&arr)[10], std::uint8_t idx) noexcept(false)
10 {
11     std::uint16_t x = arr[idx] + idx++;
12 }
13 // This will give different results depending on whether arr[idx] is evaluated
14 // before idx++ or vice versa. The problem could be avoided by putting the
15 // increment operation in a separate statement. For example:
16 void F2(std::uint8_t (&arr)[10], std::uint8_t idx) noexcept(false)
17 {
18     std::uint8_t x = arr[idx] + idx;
19     idx++;
20 }
21
22 // 2) Function arguments
23 // The order of evaluation of function arguments is unspecified.
24 extern std::uint8_t Func(std::uint8_t x, std::uint8_t y);
25 void F3() noexcept(false)
26 {
27     std::uint8_t i = 0;
28     std::uint8_t x = Func(i++, i);
29 }
30 // This will give different results depending on which of the functions two
31 // parameters is evaluated first.
32
```

```
// 3) Function pointers
// If a function is called via a function pointer there shall be no
// dependence
// on the order in which function-designator and function arguments are
// evaluated.
struct S
{
    void TaskStartFn(S* obj) noexcept(false);
};
void F4(S* p) noexcept(false)
{
    p->TaskStartFn(p++);
}

// 4) Function calls
// Functions may have additional effects when they are called (e.g. modifying
// some global data). Dependence on order of evaluation could be avoided by
// invoking the function prior to the expression that uses it, making use of a
// temporary variable for the value. For example:
extern std::uint16_t G(std::uint8_t) noexcept(false);
extern std::uint16_t Z(std::uint8_t) noexcept(false);
void F5(std::uint8_t a) noexcept(false)
{
    std::uint16_t x = G(a) + Z(a);
}
// could be written as
void F6(std::uint8_t a) noexcept(false)
{
    std::uint16_t x = G(a);
    x += Z(a);
}
// As an example of what can go wrong, consider an expression to take two values
// off a stack, subtract the second from the first, and push the result back on
// the stack:
std::int32_t Pop(std::stack<std::int32_t>& s)
{
    std::int32_t ret = s.top();
    s.pop();
    return ret;
}
void F7(std::stack<std::int32_t>& s)
{
    s.push(Pop(s) - Pop(s));
}
// This will give different results depending on which of the pop() function
// calls is evaluated first (because pop() has side effects).

// 5) Nested assignment statements
// Assignments nested within expressions cause additional side effects. The best
// way to avoid any possibility of this leading to a dependence on order of
// evaluation is not to embed assignments within expressions. For example, the
```

```
// following is not recommended:
void F8(std::int32_t& x) noexcept(false)
{
    std::int32_t y = 4;
    x = y = y++;  // It is undefined whether the final value of y is 4 or 5
}
// 6) Accessing a volatile
// The volatile type qualifier is provided in C++ to denote objects whose value
// can change independently of the execution of the program (for example an
// input register). If an object of volatile qualified type is accessed this may
// change its value. C++ compilers will not optimize out reads of a volatile. In
// addition, as far as a C++ program is concerned, a read of a volatile has a
// side effect (changing the value of the volatile). It will usually be
// necessary to access volatile data as part of an expression, which then means
// there may be dependence on order of evaluation. Where possible, though, it is
// recommended that volatiles only be accessed in simple assignment statements,
// such as the following:
void F9(std::uint16_t& x) noexcept(false)
{
    volatile std::uint16_t v;
    // ...
    x = v;
}

// The rule addresses the order of evaluation problem with side effects. Note
// that there may also be an issue with the number of times a sub-expression is
// evaluated, which is not covered by this rule. This can be a problem with
// function invocations where the function is implemented as a macro. For
// example, consider the following function-like macro and its invocation:
#define MAX(a, b) (((a) > (b)) ? (a) : (b))
// ...
void F10(std::uint32_t& i, std::uint32_t j)
{
    std::uint32_t z = MAX(i++, j);
}
// The definition evaluates the first parameter twice if a > b but only once if
// a = b. The macro invocation may thus increment i either once or twice,
// depending on the values of i and j.
// It should be noted that magnitude-dependent effects, such as those due to
// floating-point rounding, are also not addressed by this rule. Although
// the
// order in which side effects occur is undefined, the result of an operation is
// otherwise well-defined and is controlled by the structure of the expression.
// In the following example, f1 and f2 are floating-point variables; F3, F4
// and
// F5 denote expressions with floating-point types.

// f1 = F3 + ( F4 + F5 );
// f2 = ( F3 + F4 ) + F5;

// The addition operations are, or at least appear to be, performed in the order
```

```
// determined by the position of the parentheses, i.e. firstly F4 is added to F5
// then secondly F3 is added to give the value of f1. Provided that F3, F4 and
// F5 contain no side effects, their values are independent of the order in
// which they are evaluated. However, the values assigned to f1 and f2 are not
// guaranteed to be the same because floating-point rounding following the
// addition operations are dependent on the values being added.
```

See also

- MISRA C++ 2008 [7]: Rule 5-0-1 The value of an expression shall be the same under any order of evaluation that the standard permits
- HIC++ v4.0 [9]: 5.1.2: Do not rely on the sequence of evaluation within an expression.
- C++ Core Guidelines [11]: ES.40: Avoid complicated expressions
- C++ Core Guidelines [11]: ES.43: Avoid expressions with undefined order of evaluation.
- C++ Core Guidelines [11]: ES.44: Don't depend on order of evaluation of function arguments.
- C++ Core Guidelines [11]: R.13: Perform at most one explicit resource allocation in a single expression statement.

Rule M5-0-2 (advisory, implementation, partially automated)
Limited dependence should be placed on C++ operator precedence rules in expressions.

See MISRA C++ 2008 [7]

See also

- C++ Core Guidelines [11]: ES.41: If in doubt about operator precedence, parenthesize

Rule M5-0-3 (required, implementation, automated)
A cvalue expression shall not be implicitly converted to a different underlying type.

See MISRA C++ 2008 [7]

Rule M5-0-4 (required, implementation, automated)
An implicit integral conversion shall not change the signedness of the underlying type.

See MISRA C++ 2008 [7]

Rule M5-0-5 (required, implementation, automated)
There shall be no implicit floating-integral conversions.

See MISRA C++ 2008 [7]

Rule M5-0-6 (required, implementation, automated)
An implicit integral or floating-point conversion shall not reduce the size of the underlying type.

See MISRA C++ 2008 [7]

Rule M5-0-7 (required, implementation, automated)
There shall be no explicit floating-integral conversions of a cvalue expression.

See MISRA C++ 2008 [7]

Note: Standard library functions, i.e. std::floor and std::ceil, return a floating-point data type:

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

void Fn() noexcept
{
    float f = -4.5;
    std::int8_t x1 = static_cast<std::int8_t>(f);  // Compliant, x1 = -4
    std::int8_t x2 =
        static_cast<std::int8_t>(std::floor(f));  // Compliant, x2 = -5
    std::int8_t x3 =
        static_cast<std::int8_t>(std::ceil(f));  // Compliant, x3 = -4
}
```

Rule M5-0-8 (required, implementation, automated)
An explicit integral or floating-point conversion shall not increase the size of the underlying type of a cvalue expression.

See MISRA C++ 2008 [7]

Rule M5-0-9 (required, implementation, automated)
An explicit integral conversion shall not change the signedness of the underlying type of a cvalue expression.

See MISRA C++ 2008 [7]

Rule M5-0-10 (required, implementation, automated)
If the bitwise operators ~and << are applied to an operand with an underlying type of unsigned char or unsigned short, the result shall be immediately cast to the underlying type of the operand.

See MISRA C++ 2008 [7]

Rule M5-0-11 (required, implementation, automated)
The plain char type shall only be used for the storage and use of character values.

See MISRA C++ 2008 [7]

Rule M5-0-12 (required, implementation, automated)
Signed char and unsigned char type shall only be used for the storage and use of numeric values.

See MISRA C++ 2008 [7]

Rule A5-0-2 (required, implementation, automated)
The condition of an if-statement and the condition of an iteration statement shall have type bool.

Rationale

If an expression with type other than bool is used in the condition of an if-statement or iteration-statement, then its result will be implicitly converted to bool. The condition expression shall contain an explicit test (yielding a result of type bool) in order to clarify the intentions of the developer.

Note that if a type defines an explicit conversion to type bool, then it is said to be "contextually converted to bool" (Section 4.0(4) of ISO/IEC 14882:2014 [3]) and can be used as a condition of an if-statement or iteration statement.

Exception

A condition of the form type-specifier-seq declarator is not required to have type bool. This exception is introduced because alternative mechanisms for achieving the same effect are cumbersome and error-prone.

Example

```
1 // $Id: A5-0-2.cpp 289436 2017-10-04 10:45:23Z michal.szczepankiewicz $
2 #include <cstdint>
3 #include <memory>
```

```
4  extern std::int32_t* Fn();
5  extern std::int32_t Fn2();
6  extern bool Fn3();
7  void F() noexcept(false)
8  {
9      std::int32_t* ptr = nullptr;
10
11     while ((ptr = Fn()) != nullptr)  // Compliant
12     {
13         // Code
14     }
15
16     // The following is a cumbersome but compliant example
17     do
18     {
19         std::int32_t* ptr = Fn();
20
21         if (nullptr == ptr)
22         {
23             break;
24         }
25
26         // Code
27     } while (true);  // Compliant
28
29     std::unique_ptr<std::int32_t> uptr;
30     if (!uptr)  // Compliant - std::unique_ptr defines an explicit conversion to
31                 // type bool.
32     {
33         // Code
34     }
35
36     while (std::int32_t length = Fn2())  // Compliant by exception
37     {
38         // Code
39     }
40
41     while (bool flag = Fn3())  // Compliant
42     {
43         // Code
44     }
45
46     if (std::int32_t* ptr = Fn())
47         ;  // Compliant by exception
48
49     if (std::int32_t length = Fn2())
50         ;  // Compliant by exception
51
52     if (bool flag = Fn3())
53         ;  // Compliant
54
```

```
55      std::uint8_t u = 8;
56
57      if (u)
58          ;  // Non-compliant
59
60      bool boolean1 = false;
61      bool boolean2 = true;
62
63      if (u && (boolean1 <= boolean2))
64          ;  // Non-compliant
65
66      for (std::int32_t x = 10; x; --x)
67          ;  // Non-compliant
68  }
```

See also

- MISRA C++ 2008 [7]: 5-0-13 The condition of an if-statement and the condition of an iteration statement shall have type bool.

Rule M5-0-14 (required, implementation, automated)
The first operand of a conditional-operator shall have type bool.

See MISRA C++ 2008 [7]

Rule M5-0-15 (required, implementation, automated)
Array indexing shall be the only form of pointer arithmetic.

See MISRA C++ 2008 [7]

Rule M5-0-16 (required, implementation, automated)
A pointer operand and any pointer resulting from pointer arithmetic using that operand shall both address elements of the same array.

See MISRA C++ 2008 [7]

Note: The next element beyond the end of an array indicates the end of the array.

Rule M5-0-17 (required, implementation, automated)
Subtraction between pointers shall only be applied to pointers that address elements of the same array.

See MISRA C++ 2008 [7]

Rule A5-0-4 (required, implementation, automated)
Pointer arithmetic shall not be used with pointers to non-final classes.

Rationale

Pointer arithmetic is only well defined if the pointed-to type of the pointer equals the element type of the array it points into, otherwise the behavior is undefined. This property can only be guaranteed if the pointer operand is a pointer to non-class type or a pointer to final class type.

Note: This also applies to the subscripting operator as `E1[E2]` is defined in terms of pointer arithmetic as `*((E1)+(E2))`.

Example

```
// $Id: A5-0-4.cpp 309849 2018-03-02 09:36:31Z christof.meerwald $
#include <algorithm>
#include <array>
#include <cstdint>
#include <cstdlib>
#include <memory>
#include <vector>

class Base
{
  public:
    virtual ~Base() noexcept = 0;
    virtual void Do() = 0;
};

class Derived1 final : public Base
{
  public:
    void Do() final
    {
        // ...
    }

  private:
    std::int32_t m_value { 0 };
};

class Derived2 final : public Base
{
  public:
    void Do() final
    {
        // ...
    }

  private:
```

```
37      std::string m_value { };
38  };
39
40  void Foo(Base *start, size_t len)
41  {
42      // Non-Compliant: pointer arithmetic on non-final pointer type
43      for (Base *iter = start; iter != start + len; ++iter)
44      {
45          iter->Do();
46      }
47  }
48
49  void Foo(const std::vector<std::unique_ptr<Base>> &v)
50  {
51      // Compliant: uses std::unique_ptr for polymorphic objects
52      std::for_each(v.begin(), v.end(),
53          [] (const std::unique_ptr<Base> &ptr) {
54            ptr->Do();
55          });
56  }
57
58  void DoOpt(Base *obj)
59  {
60      if (obj != nullptr)
61      {
62          obj->Do();
63      }
64  }
65
66  void Bar()
67  {
68      std::array<Derived1, 2> arr1;
69      Base *base1 { arr1.data() };
70      Foo(base1, arr1.size());
71
72      DoOpt(&arr1[1]);    // Compliant: pointer arithmetic on final class
73      DoOpt(&base1[1]);   // Non-Compliant: pointer arithmetic on base class
74
75
76      std::array<Derived2, 2> arr2;
77      Base *base2 { arr2.data() };
78      Foo(base2, arr2.size());
79
80      DoOpt(arr2.data() + 1); // Compliant: pointer arithmetic on final class
81      DoOpt(base2 + 1);       // Non-Compliant: pointer arithmetic on base class
82
83
84      std::vector<std::unique_ptr<Base>> v;
85      v.push_back(std::make_unique<Derived1>());
86      v.push_back(std::make_unique<Derived2>());
87
```

```
88      Foo(v);
89  }
```

See also

- SEI CERT C++ Coding Standard [10]: CTR56-CPP: Do not use pointer arithmetic on polymorphic objects.
- JSF December 2005 [8]: AV Rule 96: Arrays shall not be treated polymorphically.
- C++ Core Guidelines [11]: T.82: Do not mix hierarchies and arrays.

Rule M5-0-18 (required, implementation, automated)
>, >=, <, <= shall not be applied to objects of pointer type, except where they point to the same array.

See MISRA C++ 2008 [7]

Rule A5-0-3 (required, implementation, automated)
The declaration of objects shall contain no more than two levels of pointer indirection.

Rationale

Use of more than two levels of indirection can seriously impair the ability to understand the behavior of the code, and therefore should be avoided.

Example

```
1  // $Id: A5-0-3.cpp 289436 2017-10-04 10:45:23Z michal.szczepankiewicz $
2  #include <cstdint>
3  using IntPtr = std::int8_t*;
4  struct S
5  {
6      std::int8_t* s1;     // Compliant
7      std::int8_t** s2;    // Compliant
8      std::int8_t*** s3;   // Non-compliant
9  };
10 S* ps1;     // Compliant
11 S** ps2;    // Compliant
12 S*** ps3;   // Non-compliant
13
14 std::int8_t** (*pfunc1)();     // Compliant
15 std::int8_t** (**pfunc2)();    // Compliant
16 std::int8_t** (***pfunc3)();   // Non-compliant
17 std::int8_t*** (**pfunc4)();   // Non-compliant
18
19 void Fn(std::int8_t* par1,          // Compliant
20         std::int8_t** par2,         // Compliant
```

```
21          std::int8_t*** par3,        // Non-compliant
22          IntPtr* par4,               // Compliant
23          IntPtr* const* const par5,  // Non-compliant
24          std::int8_t* par6[],        // Compliant
25          std::int8_t** par7[])       // Non-compliant
26  {
27      std::int8_t* ptr1;                      // Compliant
28      std::int8_t** ptr2;                     // Compliant
29      std::int8_t*** ptr3;                    // Non-compliant
30      IntPtr* ptr4;                           // Compliant
31      IntPtr* const* const ptr5 = nullptr;    // Non-compliant
32      std::int8_t* ptr6[10];                  // Compliant
33      std::int8_t** ptr7[10];                 // Compliant
34  }
35  // Explanation of types
36  // 1) par1 and ptr1 are of type pointer to std::int8_t.
37  // 2) par2 and ptr2 are of type pointer to pointer to std::int8_t.
38  // 3) par3 and ptr3 are of type pointer to a pointer to a pointer
39  // to std::int8_t.
40  // This is three levels and is non-compliant.
41  // 4) par4 and ptr4 are expanded to a type of pointer to a pointer to
42  // std::int8_t.
43  // 5) par5 and ptr5 are expanded to a type of const pointer to a const
44  // pointer
45  // to a pointer to std::int8_t. This is three levels and is non-compliant.
46  // 6) par6 is of type pointer to pointer to std::int8_t because arrays
47  // are converted
48  // to a pointer to the initial element of the array.
49  // 7) ptr6 is of type pointer to array of std::int8_t.
50  // 8) par7 is of type pointer to pointer to pointer to
51  // std::int8_t because arrays are
52  // converted to a pointer to the initial element of the array. This is
53  // three
54  // levels and is non-compliant.
55  // 9) ptr7 is of type array of pointer to pointer to std::int8_t. This
56  // is compliant.
```

See also

- MISRA C++ 2008 [7]: 5-0-19 The declaration of objects shall contain no more than two levels of pointer indirection.

Rule M5-0-20 (required, implementation, automated)
Non-constant operands to a binary bitwise operator shall have the same underlying type.

See MISRA C++ 2008 [7]

Rule M5-0-21 (required, implementation, automated)
Bitwise operators shall only be applied to operands of unsigned underlying type.

See MISRA C++ 2008 [7]

6.5.1 Primary expression

Rule A5-1-1 (required, implementation, partially automated)
Literal values shall not be used apart from type initialization, otherwise symbolic names shall be used instead.

Rationale

Avoid use of “magic” numbers and strings in expressions in preference to constant variables with meaningful names. Literal values are supposed to be used only in type initialization constructs, e.g. assignments and constructors.

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

Exception

It is allowed to use literal values in combination with logging mechanism.

Example

```
// $Id: A5-1-1.cpp 289436 2017-10-04 10:45:23Z michal.szczepankiewicz $
#include <array>
#include <cstdint>
#include <iostream>
#include <stdexcept>
namespace
{
const std::int32_t maxIterations = 10;     // Compliant - assignment
const char* const loopIterStr = "iter ";   // Compliant - assignment
const char separator = ':';                // Compliant - assignment
}
void F1() noexcept
{
    for (std::int32_t i = 0; i < 10; ++i)  // Non-compliant
    {
        std::cout << "iter " << i << ':' << '\n';  // Compliant by exception
    }

    for (std::int32_t i = 0; i < maxIterations; ++i)  // Compliant
    {
        std::cout << loopIterStr << i << separator << '\n';  // Compliant
```

```
    }

    for (std::int32_t i = 0; i < maxIterations; ++i)  // Compliant
    {
        std::cout << "iter " << i << ':' << '\n';  // Compliant by exception
    }
}
void F2()
{
    // ...
    throw std::logic_error("Logic Error");  // Compliant
    // initialization of exception object
}
class C
{
  public:
    C() : x(0), y(nullptr)  // Compliant - initialization
    {
    }
    C(std::int8_t num, std::int32_t* ptr) : x(num), y(ptr) {}

  private:
    std::int8_t x;
    std::int32_t* y;
};
static std::int32_t* globalPointer = nullptr;  // Compliant - assignment
void F3() noexcept
{
    C c1;
    // ...
    C c2(0, globalPointer);  // Compliant - initialization of C object
}
std::int32_t F4(std::int32_t x, std::int32_t y) noexcept
{
    return x + y;
}
void F5() noexcept
{
    std::int32_t ret = F4(2, 5);  // Non-compliant
    // ...
    std::int32_t x = 2;
    std::int32_t y = 5;
    ret = F4(x, y);  // Compliant

    std::array<std::int8_t, 5> arr{{1, 2, 3, 4, 5}};  // Compliant
}
```

See also

- HIC++ v4.0 [9]: 5.1.1 Use symbolic names instead of literal values in code.

Rule A5-1-2 (required, implementation, automated)
Variables shall not be implicitly captured in a lambda expression.

Rationale

Capturing variables explicitly helps document the intention of the author. It also allows for different variables to be explicitly captured by copy or by reference within the lambda definition.

Exception

It is allowed to implicitly capture variables with non-automatic storage duration.

Example

```
// $Id: A5-1-2.cpp 289436 2017-10-04 10:45:23Z michal.szczepankiewicz $
#include <algorithm>
#include <cstdint>
#include <vector>
void Fn1(const std::vector<std::int32_t>& v)
{
    std::uint64_t sum = 0;
    std::for_each(v.begin(), v.end(), [&](std::int32_t lhs) {
        sum += lhs;
    });  // Non-compliant

    sum = 0;
    std::for_each(v.begin(), v.end(), [&sum](std::int32_t lhs) {
        sum += lhs;
    });  // Compliant
}
void Fn2()
{
    constexpr std::uint8_t n = 10;
    static std::int32_t j = 0;
    [n]() {
        std::int32_t array[n];  // Compliant
        j += 1;                 // Compliant by exception
    };
}
```

See also

- HIC++ v4.0 [9]: 5.1.4 Do not capture variables implicitly in a lambda.
- C++ Core Guidelines [11]: F.54: If you capture this, capture all variables explicitly (no default capture).

Rule A5-1-3 (required, implementation, automated)
Parameter list (possibly empty) shall be included in every lambda expression.

Rationale

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.

Example

```
// $Id: A5-1-3.cpp 289436 2017-10-04 10:45:23Z michal.szczepankiewicz $
#include <cstdint>
void Fn()
{
    std::int32_t i = 0;
    std::int32_t j = 0;
    auto lambda1 = [&i, &j] { ++i, ++j; };  // Non-compliant
    auto lambda2 = [&i, &j]() {
        ++i;
        ++j;
    };  // Compliant
}
```

See also

- HIC++ v4.0 [9]: 5.1.5 Include a (possibly empty) parameter list in every lambda expression

Rule A5-1-4 (required, implementation, automated)
A lambda expression object shall not outlive any of its reference-captured objects.

Rationale

When an object is captured by reference in a lambda, lifetime of the object is not tied to the lifetime of the lambda.

If a lambda object leaves the scope of one of its reference-captured object, the execution of the lambda expression results in an undefined behavior once the reference-captured object is accessed.

Example

```
// $Id: A5-1-4.cpp 289436 2017-10-04 10:45:23Z michal.szczepankiewicz $
#include <cstdint>
```

```
#include <functional>
std::function<std::int32_t()> F()
{
    std::int32_t i = 12;
    return ([&i]() -> std::int32_t {
        i = 100;
        return i;
    });  // Non-compliant
}
std::function<std::int32_t()> G()
{
    std::int32_t i = 12;
    return ([i]() mutable -> std::int32_t { return ++i; });  // Compliant
}
void Fn()
{
    auto lambda1 = F();
    std::int32_t i = lambda1();  // Undefined behavior
    auto lambda2 = G();
    i = lambda2();  // lambda2() returns 13
}
```

See also

- SEI CERT C++ [10]: EXP61-CPP. A lambda object must not outlive any of its reference captured objects.
- C++ Core Guidelines [11]: F.53: Avoid capturing by reference in lambdas that will be used nonlocally, including returned, stored on the heap, or passed to another thread.

Rule A5-1-6 (advisory, implementation, automated)
Return type of a non-void return type lambda expression should be explicitly specified.

Rationale

If a non-void return type lambda expression does not specify its return type, then it may be confusing which type it returns. It leads to developers confusion.

Note that, while the return type is specified, implicit conversion between type of returned value and return type specified in the lambda expression may occur. This problem should not be ignored.

Example

```
// $Id: A5-1-6.cpp 289436 2017-10-04 10:45:23Z michal.szczepankiewicz $
#include <cstdint>
void Fn() noexcept
{
```

```
    auto lambda1 = []() -> std::uint8_t {
        std::uint8_t ret = 0U;
        // ...
        return ret;
    };  // Compliant
    auto lambda2 = []() {
        // ...
        return 0U;
    };                    // Non-compliant - returned type is not specified
    auto x = lambda1();   // Type of x is std::uint8_t
    auto y = lambda2();   // What is the type of y?
}
```

See also

- none

Rule A5-1-7 (required, implementation, automated)
A lambda shall not be an operand to decltype or typeid.

Rationale

"The type of the lambda-expression (which is also the type of the closure object) is a unique, unnamed non-union class type [...]" [C++14 Language Standard] [3]

Each lambda expression has a different unique underlying type, and therefore the type is not to be used as a decltype or typeid argument. It is allowed to use it as a template parameter and a function argument.

Example

```
// $Id: A5-1-7.cpp 289815 2017-10-06 11:19:11Z michal.szczepankiewicz $
#include <cstdint>
#include <functional>
#include <vector>
void Fn()
{
    auto lambda1 = []() -> std::int8_t { return 1; };
    auto lambda2 = []() -> std::int8_t { return 1; };

    if (typeid(lambda1) == typeid(lambda2))  // Non-compliant - types of lambda1
                                             // and lambda2 are different
    {
        // ...
    }

    std::vector<decltype(lambda1)> v;  // Non-compliant
    // v.push_back([]() { return 1; });  // Compilation error, type of pushed
    // lambda is different than decltype(lambda1)
    // using mylambda_t = decltype([](){ return 1; });  // Non-compliant -
    // compilation error
```

```
    auto lambda3 = [] () { return 2; };
    using lambda3_t = decltype(lambda3);  // Non-compliant - lambda3_t type can
                                          // not be used for lambda expression
                                          // declarations
    // lambda3_t lambda4 = [] () { return 2; };  // Conversion error at
    // compile-time
    std::function<std::int32_t()> f1 = [] () { return 3; };
    std::function<std::int32_t()> f2 = [] () { return 3; };

    if (typeid(f1) == typeid(f2))  // Compliant - types are equal
    {
        // ...
    }
}

template <typename T>
void Foo(T t);

void Bar()
{
    Foo([] () {});  // Compliant
}
```

See also

- none

Rule A5-1-8 (advisory, implementation, automated)
Lambda expressions should not be defined inside another lambda expression.

Rationale

Defining lambda expressions inside other lambda expressions reduces readability of the code.

Example

```
// $Id: A5-1-8.cpp 289436 2017-10-04 10:45:23Z michal.szczepankiewicz $
#include <cstdint>
void Fn1()
{
    std::int16_t x = 0;
    auto f1 = [&x] () {

        auto f2 = [] () {};  // Non-compliant
        f2();

        auto f4 = [] () {};  // Non-compliant
        f4();
```

```
13
14      };  // Non-compliant
15
16      f1();
17  }
18  void Fn2()
19  {
20      auto f5 = []() {
21          // Implementation
22      };  // Compliant
23      f5();
24  }
```

See also

- none

Rule A5-1-9 (advisory, implementation, automated)
Identical unnamed lambda expressions shall be replaced with a named function or a named lambda expression.

Rationale

Code duplication reduces readability and maintainability as it might not be obvious that the lambda expressions are identical and any changes need to be applied in more than one place.

Example

```
1   // $Id: A5-1-9.cpp 307019 2018-02-09 15:16:47Z christof.meerwald $
2   #include <algorithm>
3   #include <cstdint>
4   #include <vector>
5
6   void Fn1(const std::vector<int16_t> &v)
7   {
8       // Non-compliant: identical unnamed lambda expression
9       if (std::none_of(v.begin(), v.end(),
10                  [] (int16_t i) { return i < 0; }))
11      {
12          // ...
13      }
14      else if (std::all_of(v.begin(), v.end(),
15                  [] (int16_t i) { return i < 0; }))
16      {
17          // ...
18      }
19  }
20
21  void Fn2(const std::vector<int16_t> &v)
```

```
{
    // Compliant: re-using lambda expression
    auto is_negative = [] (int16_t i) { return i < 0; };

    if (std::none_of(v.begin(), v.end(), is_negative))
    {
        // ...
    }
    else if (std::all_of(v.begin(), v.end(), is_negative))
    {
        // ...
    }
}
```

See also

- C++ Core Guidelines [11]: T.141: Use an unnamed lambda if you need a simple function object in one place only.

6.5.2 Postfix expressions

Rule M5-2-2 (required, implementation, automated)
A pointer to a virtual base class shall only be cast to a pointer to a derived class by means of dynamic_cast.

See MISRA C++ 2008 [7]

See also

- JSF December 2005 [8]: AV Rule 178: Down casting (casting from base to derived class) shall only be allowed through one of the following mechanism: Virtual functions that act like dynamic casts (most likely useful in relatively simple cases); Use of the visitor (or similar) pattern (most likely useful in complicated cases).

Rule M5-2-3 (advisory, implementation, automated)
Casts from a base class to a derived class should not be performed on polymorphic types.

See MISRA C++ 2008 [7]

Note: Type is polymorphic if it declares or inherits at least one virtual function.

Rule A5-2-1 (advisory, implementation, automated)
dynamic_cast should not be used.

Rationale

Implementations of dynamic_cast mechanism are unsuitable for use with real-time systems where low memory usage and determined performance are essential.

If dynamic casting is essential for your program, usage of its custom implementation should be considered. Also, usage of the dynamic_cast can be replaced with polymorphism, i.e. virtual functions.

Example

```
1  // $Id: A5-2-1.cpp 289436 2017-10-04 10:45:23Z michal.szczepankiewicz $
2  class A
3  {
4    public:
5      virtual void F() noexcept;
6  };
7  class B : public A
8  {
9    public:
10     void F() noexcept override {}
11 };
12 void Fn(A* aptr) noexcept
13 {
14     // ...
15     B* bptr = dynamic_cast<B*>(aptr);  // Non-compliant
16
17     if (bptr != nullptr)
18     {
19         // Use B class interface
20     }
21     else
22     {
23         // Use A class interface
24     }
25 }
```

See also

- JSF December 2005 [8]: AV Rule 178: Down casting (casting from base to derived class) shall only be allowed through one of the following mechanism: Virtual functions that act like dynamic casts (most likely useful in relatively simple cases); Use of the visitor (or similar) pattern (most likely useful in complicated cases).
- C++ Core Guidelines [11]: C.146: Use dynamic_cast where class hierarchy navigation is unavoidable.

- Journal of Computing Science and Engineering, Damian Dechev, Rabi Mahapatra, Bjarne Stroustrup: Practical and Verifiable C++ Dynamic Cast for Hard Real-Time Systems.
- Software-Practice and Experience, Michael Gibbs and Bjarne Stroustrup: Fast dynamic casting.

Rule A5-2-2 (required, implementation, automated)
Traditional C-style casts shall not be used.

Rationale

C-style casts are more dangerous than the C++ named conversion operators. The C-style casts are difficult to locate in large programs and the intent of the conversion is not explicit.

Traditional C-style casts raise several concerns:

- C-style casts enable most any type to be converted to most any other type without any indication of the reason for the conversion
- C-style cast syntax is difficult to identify for both reviewers and tools. Consequently, both the location of conversion expressions as well as the subsequent analysis of the conversion rationale proves difficult for C-style casts

Thus, C++ introduces casts (const_cast, dynamic_cast, reinterpret_cast, and static_cast) that address these problems. These casts are not only easy to identify, but they also explicitly communicate the developer's intent for applying a cast.

Example

```
1  // $Id: A5-2-2.cpp 289436 2017-10-04 10:45:23Z michal.szczepankiewicz $
2  #include <cstdint>
3  class C
4  {
5    public:
6      explicit C(std::int32_t) {}
7      virtual void Fn() noexcept {}
8  };
9  class D : public C
10 {
11   public:
12     void Fn() noexcept override {}
13 };
14 class E
15 {
16 };
17 std::int32_t G() noexcept
18 {
19     return 7;
20 }
```

```
void F() noexcept(false)
{
    C a1 = C{10};        // Compliant
    C* a2 = (C*)(&a1);   // Non-compliant
    const C a3(5);
    C* a4 = const_cast<C*>(&a3);       // Compliant - violates another rule
    E* d1 = reinterpret_cast<E*>(a4);  // Compliant - violates another rule
    D* d2 = dynamic_cast<D*>(a2);      // Compliant - violates another rule
    std::int16_t x1 = 20;
    std::int32_t x2 = static_cast<std::int32_t>(x1);  // Compliant
    std::int32_t x3 = (std::int32_t)x1;               // Non-compliant
    std::int32_t x4 = 10;
    float f1 = static_cast<float>(x4);                // Compliant
    float f2 = (float)x4;                             // Non-compliant
    std::int32_t x5 = static_cast<std::int32_t>(f1);  // Compliant
    std::int32_t x6 = (std::int32_t)f1;               // Non-compliant
    (void)G();                                        // Non-compliant
    static_cast<void>(G());                           // Compliant
}
```

See also

- MISRA C++ 2008 [7]: 5-2-4 C-style casts (other than void casts) and functional notation casts (other than explicit constructor calls) shall not be used.
- JSF December 2005 [8]: AV Rule 185 C++ style casts (const_cast, reinterpret_cast, and static_cast) shall be used instead of the traditional C-style casts.

Rule A5-2-3 (required, implementation, automated)
A cast shall not remove any const or volatile qualification from the type of a pointer or reference.

Rationale

Removal of the const or volatile qualification may not meet developer expectations as it may lead to undefined behavior.

Note that either const_cast and traditional C-style casts that remove const or volatile qualification shall not be used.

Example

```
// $Id: A5-2-3.cpp 289436 2017-10-04 10:45:23Z michal.szczepankiewicz $
#include <cstdint>
void F1(const char* str) noexcept(false)
{
    *(const_cast<char*>(str)) =
        '\0';  // Non-compliant - const qualification removed
}
```

```
class C
{
  public:
    explicit C(std::int32_t) {}
};
void F2() noexcept(false)
{
    C const a1 = C(10);
    C* a2 = const_cast<C*>(&a1);  // Non-compliant - const qualification removed
    C* a3 = (C*)&a1;              // Non-compliant - const qualification removed
}
extern volatile std::int32_t* F3() noexcept;
void F4() noexcept
{
    volatile std::int32_t* ptr1 = F3();
    // ...
    std::int32_t* ptr2 = const_cast<std::int32_t*>(
        ptr1);  // Non-compliant - volatile qualification removed
    // ...
    std::int32_t* ptr3 =
        (std::int32_t*)ptr1;  // Non-compliant - volatile qualification removed
}
```

See also

- MISRA C++ 2008 [7]: 5-2-5 A cast shall not remove any const or volatile qualification from the type of a pointer or reference.

Rule M5-2-6 (required, implementation, automated)
A cast shall not convert a pointer to a function to any other pointer type, including a pointer to function type.

See MISRA C++ 2008 [7]

Rule A5-2-4 (required, implementation, automated)
reinterpret_cast shall not be used.

Rationale

Use of reinterpret_cast may violate type safety and cause the program to access a variable as if it were of another, unrelated type.

Example

```
// $Id: A5-2-4.cpp 289436 2017-10-04 10:45:23Z michal.szczepankiewicz $
#include <cstdint>
#include <string>
void F1() noexcept
```

```
{
    std::string str = "Hello";
    std::int32_t* ptr = reinterpret_cast<std::int32_t*>(&str);  // Non-compliant
}
struct A
{
    std::int32_t x;
    std::int32_t y;
};
class B
{
  public:
    virtual ~B() {}

  private:
    std::int32_t x;
};
class C : public B
{
};
class D : public B
{
};
void F2(A* ptr) noexcept
{
    B* b1 = reinterpret_cast<B*>(ptr);  // Non-compliant
    std::int32_t num = 0;
    A* a1 = reinterpret_cast<A*>(num);  // Non-compliant
    A* a2 = (A*)
        num;  // Compliant with this rule, but non-compliant with Rule A5-2-2.
    B* b2 = reinterpret_cast<B*>(num);  // Non-compliant
    D d;
    C* c1 = reinterpret_cast<C*>(&d);  // Non-compliant - cross cast
    C* c2 = (C*)&d;  // Compliant with this rule, but non-compliant with Rule
                     // A5-2-2. Cross-cast.
    B* b3 = &d;      // Compliant - class D is a subclass of class B
}
```

See also

- MISRA C++ 2008 [7]: Rule 5-2-7 An object with pointer type shall not be converted to an unrelated pointer type, either directly or indirectly.
- C++ Core Guidelines [11]: Type.1: Don't use reinterpret_cast.

Rule A5-2-6 (required, implementation, automated)
The operands of a logical `&&` or `\\` shall be parenthesized if the operands contain binary operators.

Rationale

Parentheses are required to add clarity in logical expressions making code easier to review versus code based only C++ operator precedence rules.

Example

```
// $Id: A5-2-6.cpp$

#include <cstdint>

void Fn(std::int32_t value) noexcept
{
    if (value > 0 && value < 3) // Non-compliant
    {
        // do some work
    }
    else if ((value > 1) && (value < 2)) // Compliant
    {
        // do some work
    }
    else
    {
        // do some work
    }

    return;
}
```

See also

- MISRA C++ 2008 [7]: M5-2-1: Each operand of a logical && or || shall be a postfix expression.
- JSF December 2005 [8]: AV Rule 158: The operands of a logical `&&` or `\\` shall be parenthesized if the operands contain binary operators.
- C++ Core Guidelines [11]: ES.41: If in doubt about operator precedence, parenthesize

Rule M5-2-8 (required, implementation, automated)
An object with integer type or pointer to void type shall not be converted to an object with pointer type.

See MISRA C++ 2008 [7]

Rule M5-2-9 (required, implementation, automated)
A cast shall not convert a pointer type to an integral type.

See MISRA C++ 2008 [7]

Note: Obligation level changed.

Rule M5-2-10 (required, implementation, automated)
The increment (++) and decrement (--) operators shall not be mixed with other operators in an expression.

See MISRA C++ 2008 [7]

Note: Obligation level changed.

Rule M5-2-11 (required, implementation, automated)
The comma operator, `&&` operator and `the ||` operator shall not be overloaded.

See MISRA C++ 2008 [7]

Rule A5-2-5 (required, implementation, automated)
An array or container shall not be accessed beyond its range.

Rationale

To avoid undefined behavior, range checks should be coded to ensure that container access via iterator arithmetic or subscript operator is within defined bounds. This could also be achieved by accessing an array via a subscript operator with constant indices only.

When copying data via standard library algorithms (such as `std::copy` or `std::transform`), the target destination must be guaranteed to be large enough to hold the data.

Note: This rule applies to C-style arrays and all other containers (including `std::basic_string`) that access their elements via iterators or via an index. The term iterator includes pointers.

Note: Calculating an iterator one past the last element of the array is well defined, but dereferencing such an iterator is not.

Example

```
1 // $Id: A5-2-5.cpp 289436 2017-10-04 10:45:23Z michal.szczepankiewicz $
2 #include <array>
3 #include <cstdint>
4 #include <iostream>
5 void Fn1() noexcept
6 {
```

```
7       constexpr std::int32_t arraySize = 16;
8       std::int32_t array[arraySize]{0};
9
10      std::int32_t elem1 =
11          array[0];  // Compliant - access with constant literal that
12                     // is less than ArraySize
13      std::int32_t elem2 =
14          array[12];  // Compliant - access with constant literal that
15                      // is less than ArraySize
16      for (std::int32_t idx = 0; idx < 20; ++idx)
17      {
18          std::int32_t elem3 =
19              array[idx];  // Non-compliant - access beyond ArraySize
20                           // bounds, which has 16 elements
21      }
22
23      std::int32_t shift = 25;
24      std::int32_t elem4 =
25          *(array + shift);  // Non-compliant - access beyond ArraySize bounds
26
27      std::int32_t index = 0;
28      std::cin >> index;
29      std::int32_t elem5 =
30          array[index];  // Non-compliant - index may exceed the ArraySize bounds
31      if (index < arraySize)
32      {
33          std::int32_t elem6 = array[index];  // Compliant - range check coded
34      }
35  }
36  void Fn2() noexcept
37  {
38      constexpr std::int32_t arraySize = 32;
39      std::array<std::int32_t, arraySize> array;
40      array.fill(0);
41
42      std::int32_t elem1 =
43          array[10];  // Compliant - access with constant literal that
44                      // is less than ArraySize
45      std::int32_t index = 40;
46      std::int32_t elem2 =
47          array[index];  // Non-compliant - access beyond ArraySize bounds
48      try
49      {
50          std::int32_t elem3 =
51              array.at(50);  // Compliant - at() method provides a
52                             // range check, throwing an exception if
53                             // input exceeds the bounds
54      }
55      catch (std::out_of_range&)
56      {
57          // Handle an error
```

```
58      }
59
60      for (auto&& e : array)  // The std::array provides a possibility to iterate
61                              // over its elements with range-based loop
62      {
63          // Iterate over all elements
64      }
65  }
```

See also

- HIC++ v4.0 [9]: 5.2.1: Ensure that pointer or array access is demonstrably within bounds of a valid object.

Rule M5-2-12 (required, implementation, automated)
An identifier with array type passed as a function argument shall not decay to a pointer.

See MISRA C++ 2008 [7]

See also

- C++ Core Guidelines [11]: C.152: Never assign a pointer to an array of derived class objects to a pointer to its base.
- C++ Core Guidelines [11]: R.2: In interfaces, use raw pointers to denote individual objects (only).
- C++ Core Guidelines [11]: I.13: Do not pass an array as a single pointer.

6.5.3 Unary expressions

Rule M5-3-1 (required, implementation, automated)
Each operand of the ! operator, the logical `&&` or `the logical ||` operators shall have type bool.

See MISRA C++ 2008 [7]

Rule M5-3-2 (required, implementation, automated)
The unary minus operator shall not be applied to an expression whose underlying type is unsigned.

See MISRA C++ 2008 [7]

Rule M5-3-3 (required, implementation, automated)
The unary & operator shall not be overloaded.

See MISRA C++ 2008 [7]

Rule M5-3-4 (required, implementation, automated)
Evaluation of the operand to the sizeof operator shall not contain side effects.

See MISRA C++ 2008 [7]

Rule A5-3-1 (required, implementation, non-automated)
Evaluation of the operand to the typeid operator shall not contain side effects.

Rationale

The operand of typeid operator is evaluated only if it is a function call which returns a reference to a polymorphic type.

Providing side effects to typeid operator, which takes place only under special circumstances, makes the code more difficult to maintain.

Example

```
1  // $Id: A5-3-1.cpp 289436 2017-10-04 10:45:23Z michal.szczepankiewicz $
2  #include <typeinfo>
3  bool SideEffects() noexcept
4  {
5      // Implementation
6      return true;
7  }
8  class A
9  {
10   public:
11     static A& F1() noexcept { return a; }
12     virtual ~A() {}
13
14   private:
15     static A a;
16 };
17 A A::a;
18 void F2() noexcept(false)
19 {
20     typeid(SideEffects());  // Non-compliant - sideEffects() function not called
21     typeid(A::F1());  // Non-compliant - A::f1() functions called to determine
22                       // the polymorphic type
23 }
```

See also

- HIC++ v4.0 [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.

Rule A5-3-2 (required, implementation, partially automated)
Null pointers shall not be dereferenced.

Rationale

Dereferencing a NULL pointer leads to undefined behavior.

Note: It is required requires that a pointer is checked for non-NULL status before dereferencing occurs.

Example

```
// $Id: A5-3-2.cpp 305629 2018-01-29 13:29:25Z piotr.serwa $
#include <iostream>
#include <memory>
#include <cstdint>

class A
{
public:
    A(std::uint32_t a) : a(a) {}
    std::uint32_t GetA() const noexcept { return a; }

private:
    std::uint32_t a;
};

bool Sum(const A* lhs, const A* rhs)
{
    //non-compliant, not checking if pointer is invalid
    return lhs->GetA() + rhs->GetA();
}


int main(void)
{
    auto l = std::make_shared<A>(3);
    decltype(l) r;

    auto sum = Sum(l.get(), r.get());

    std::cout << sum << std::endl;
    return 0;
}
```

See also

- JSF December 2005 [8]: AV Rule 174: The null pointer shall not be dereferenced.
- SEI CERT C++ Coding Standard [10]: EXP34-C: Do not dereference null pointers.
- C++ Core Guidelines [11]: ES.65: Don't dereference an invalid pointer.

Rule A5-3-3 (required, implementation, automated)
Pointers to incomplete class types shall not be deleted.

Rationale

Incomplete class types are forward declared class types, for which the compiler has not yet seen a definition. It is undefined behavior if a pointer to an incomplete class type is deleted, and the class has a non-trivial destructor or a deallocation function. This rule prohibits deletion of such a pointer even in the harmless case of a trivially destructible class type without a deallocation function, since a non-trivial destructor or a deallocation function may be added later as the code evolves.

Example

```
1  // $Id: A5-3-3.cpp 309184 2018-02-26 20:38:28Z jan.babst $
2
3  // Non-compliant: At the point of deletion, pimpl points
4  // to an incomplete class type.
5  class Bad
6  {
7      class Impl;
8      Impl* pimpl;
9
10   public:
11     // ...
12     ~Bad() { delete pimpl; }  // violates A18-5-2
13 };
14
15 // Compliant: At the point of deletion, pimpl points to
16 // a complete class type.
17
18 // In a header file ...
19 class Good
20 {
21     class Impl;
22     Impl* pimpl;
23
24   public:
25     // ...
26     ~Good();
27 };
28
```

```
// In an implementation file ...
class Good::Impl
{
    // ...
};
// Good::Impl is a complete type now

Good::~Good()
{
    delete pimpl;  // violates A18-5-2
}

// Compliant: Contemporary solution using std::unique_ptr
// and conforming to A18-5-2.
// Note that std::unique_ptr<Impl> requires Impl to be a complete type
// at the point where pimpl is deleted and thus automatically enforces
// A5-3-3. This is the reason why the destructor of Better must be defined in an
// implementation file when Better::Impl is a complete type, even if the
// definition is just the default one.

// In a header file ...
#include <memory>
class Better
{
    class Impl;
    std::unique_ptr<Impl> pimpl;

  public:
    // ...
    ~Better();
};

// In an implementation file ...
class Better::Impl
{
    // ...
};
// Better::Impl is a complete type now

Better::~Better() = default;
```

See also

- ISO/IEC 14882:2014 [3]: 5.3.5: [expr.delete]
- SEI CERT C++ Coding Standard [10]: EXP57-CPP: Do not cast or delete pointers to incomplete classes.

6.5.5 Pointer-to-member

Rule A5-5-1 (required, implementation, automated)
A pointer to member shall not access non-existent class members.

Rationale

Usage of a pointer-to-member expression leads to undefined behavior in the following cases:

- The pointer to member is a null pointer.
- The dynamic type of the object does not contain the member to which the called pointer to member refers.

Example

```
// $Id: A5-5-1.cpp 302200 2017-12-20 17:17:08Z michal.szczepankiewicz $

class A
{
public:
    virtual ~A() = default;
};

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

    using ptr = void (AA::*)();
};

class B
{
public:
    static AA::ptr foo_ptr2;
};

AA::ptr B::foo_ptr2;

int main(void)
{
    A* a = new A();
    void (A::*foo_ptr1)() = static_cast<void(A::*)()>(&AA::foo);
    (a->*foo_ptr1)(); // non-compliant
    delete a;

    AA* aa = new AA();
    (aa->*B::foo_ptr2)(); // non-compliant, not explicitly initialized
    delete aa;

    return 0;
```

```
38 }
```

See also

- SEI CERT C++ Coding Standard [10]: OOP55-CPP: Do not use pointer-to-member operators to access nonexistent members

6.5.6 Multiplicative operators

Rule A5-6-1 (required, implementation, automated)
The right hand operand of the integer division or remainder operators shall not be equal to zero.

Rationale

The result is undefined if the right hand operand of the integer division or the remainder operator is zero.

Example

```
1  // $Id: A5-6-1.cpp 305629 2018-01-29 13:29:25Z piotr.serwa $
2  #include <cstdint>
3  #include <stdexcept>
4  std::int32_t Division1(std::int32_t a, std::int32_t b) noexcept
5  {
6      return (a / b);  // Non-compliant - value of b could be zero
7  }
8  std::int32_t Division2(std::int32_t a, std::int32_t b)
9  {
10     if (b == 0)
11     {
12         throw std::runtime_error("Division by zero error");
13     }
14     return (a / b);  // Compliant - value of b checked before division
15 }
16 double Fn()
17 {
18     std::int32_t x = 20 / 0;  // Non-compliant - undefined behavior
19     x = Division1(20, 0);     // Undefined behavior
20     x = Division2(20,
21                   0);  // Preconditions check will throw a runtime_error from
22                        // division2() function
23     std::int32_t remainder = 20 % 0;  // Non-compliant - undefined behavior
24 }
```

See also

- HIC++ v4.0 [9]: 5.5.1 Ensure that the right hand operand of the division or remainder operators is demonstrably non-zero.

- C++ Core Guidelines [11]: ES.105: Don't divide by zero.

6.5.8 Shift operators

Rule M5-8-1 (required, implementation, partially automated)
The right hand operand of a shift operator shall lie between zero and one less than the width in bits of the underlying type of the left hand operand.

See MISRA C++ 2008 [7]

6.5.10 Equality operators

Rule A5-10-1 (required, implementation, automated)
A pointer to member virtual function shall only be tested for equality with null-pointer-constant.

Rationale

The result of equality comparison between pointer to member virtual function and anything other than null-pointer-constant (i.e. nullptr, see: A4-10-1) is unspecified.

Example

```
1  // $Id: A5-10-1.cpp 289436 2017-10-04 10:45:23Z michal.szczepankiewicz $
2  class A
3  {
4    public:
5      virtual ~A() = default;
6      void F1() noexcept {}
7      void F2() noexcept {}
8      virtual void F3() noexcept {}
9  };
10
11 void Fn()
12 {
13     bool b1 = (&A::F1 == &A::F2);   // Compliant
14     bool b2 = (&A::F1 == nullptr);  // Compliant
15     bool b3 = (&A::F3 == nullptr);  // Compliant
16     bool b4 = (&A::F3 != nullptr);  // Compliant
17     bool b5 = (&A::F3 == &A::F1);   // Non-compliant
18 }
```

See also

- HIC++ v4.0 [9]: 5.7.2 Ensure that a pointer to member that is a virtual function is only compared (==) with nullptr.

- JSF December 2005 [8]: AV Rule 97.1 Neither operand of an equality operator (== or !=) shall be a pointer to a virtual member function.

6.5.14 Logical AND operator

Rule M5-14-1 (required, implementation, automated)
The right hand operand of a logical `&&`, `||` operators shall not contain side effects.

See MISRA C++ 2008 [7]

6.5.16 Conditional operator

Rule A5-16-1 (required, implementation, automated)
The ternary conditional operator shall not be used as a sub-expression.

Rationale

Using the result of the ternary conditional operator as an operand or nesting conditional operators makes the code less readable and more difficult to maintain.

Example

```
1  // $Id: A5-16-1.cpp 289436 2017-10-04 10:45:23Z michal.szczepankiewicz $
2  #include <cstdint>
3  constexpr bool Fn1(std::int32_t x)
4  {
5      return (x > 0);  // Compliant
6  }
7  std::int32_t Fn2(std::int32_t x)
8  {
9      std::int32_t i = (x >= 0 ? x : 0);  // Compliant
10
11     std::int32_t j =
12         x + (i == 0 ? (x >= 0 ? x : -x) : i);  // Non-compliant - nested
13                                                // conditional operator
14                                                // and used as a
15                                                // sub-expression
16     return (
17         i > 0
18             ? (j > 0 ? i + j : i)
19             : (j > 0 ? j : 0));  // Non-compliant - nested conditional operator
20 }
```

See also

- HIC++ v4.0 [9]: 5.8.1 Do not use the conditional operator (?:) as a sub-expression.

6.5.18 Assignment and compound assignment operation

Rule M5-17-1 (required, implementation, non-automated)
The semantic equivalence between a binary operator and its assignment operator form shall be preserved.

See MISRA C++ 2008 [7]

6.5.19 Comma operator

Rule M5-18-1 (required, implementation, automated)
The comma operator shall not be used.

See MISRA C++ 2008 [7]

6.5.20 Constant expression

Rule M5-19-1 (required, implementation, automated)
Evaluation of constant unsigned integer expressions shall not lead to wrap-around.

See MISRA C++ 2008 [7]

Note: Obligation level changed

Note: This rule applies to bit-fields, too.

6.6 Statements

6.6.2 Expression statement

Rule M6-2-1 (required, implementation, automated)
Assignment operators shall not be used in sub-expressions.

See MISRA C++ 2008 [7]

Exception

It is allowed that a condition of the form type-specifier-seq declarator uses an assignment operator. This exception is introduced because alternative mechanisms for achieving the same effect are cumbersome and error-prone.

Rule A6-2-1 (required, implementation, automated)
Move and copy assignment operators shall either move or respectively copy base classes and data members of a class, without any side effects.

Rationale

It is expected behavior that the move/copy assigned operator are only used to move/copy the object of the class type and possibly set moved-from object to a valid state. Those operators are not supposed to provide any performance overhead or side effects that could affect moving or copying the object.

Note: Class members that are not essential for a class invariant may not need to be moved/copied (e.g. caches, debug information).

Example

```
// $Id: A6-2-1.cpp 305786 2018-01-30 08:58:33Z michal.szczepankiewicz $

#include <cstdint>
#include <utility>
class A
{
public:
    A& operator=(const A& oth)  // Compliant
    {
        if(&oth == this)
        {
            return *this;
        }
        x = oth.x;
        return *this;
    }

  private:
    std::int32_t x;
};
class B
{
public:
    ~B() { delete ptr; }

    //compliant
    B& operator=(B&& oth)
    {
```

```
29          if(&oth == this)
30          {
31              return *this;
32          }
33          ptr = std::move(oth.ptr);
34          // Compliant - this is not a side-effect, in this
35          // case it is essential to leave moved-from object
36          // in a valid state, otherwise double deletion will
37          // occur.
38          return *this;
39      }
40
41  private:
42      std::int32_t* ptr;
43  };
44
45  class C
46  {
47  public:
48      C& operator=(const C& oth)
49      {
50          if(&oth == this)
51          {
52              return *this;
53          }
54          x = oth.x % 2;  // Non-compliant - unrelated side-effect
55          return *this;
56      }
57
58  private:
59      std::int32_t x;
60  };
61
62  class D
63  {
64  public:
65      explicit D(std::uint32_t a) : a(a), noOfModifications(0) {}
66
67      D& operator=(const D& oth)
68      {
69          if(&oth == this)
70          {
71              return *this;
72          }
73          a = oth.a;
74          //compliant, not copying the debug information about number of
        modifications
75          return *this;
76      }
77
78      void SetA(std::uint32_t aa)
```

```
79      {
80          ++noOfModifications;
81          a = aa;
82      }
83      std::uint32_t GetA() const noexcept
84      {
85          return a;
86      }
87
88  private:
89      std::uint32_t a;
90      std::uint64_t noOfModifications;
91  };
```

See also

- JSF December 2005 [8]: AV Rule 83: An assignment operator shall assign all data members and bases that affect the class invariant (a data element representing a cache, for example, would not need to be copied).

Rule A6-2-2 (required, implementation, automated)
Expression statements shall not be explicit calls to constructors of temporary objects only.

Rationale

The developer's intention might have been to define an unnamed local variable that would live until the end of the scope to implement the RAII pattern (Resource Acquisition Is Initialization). But as there are no unnamed variables in C++, it is in fact only creating a temporary object that gets destroyed again at the end of the full expression.

Example

```
1   // $Id: A6-2-2.cpp 326655 2018-07-20 14:58:55Z christof.meerwald $
2
3   #include <cstdint>
4   #include <fstream>
5   #include <mutex>
6
7   class A
8   {
9   public:
10      void SetValue1(std::int32_t value)
11      {
12          std::lock_guard<std::mutex> {m_mtx}; // Non-compliant: temporary object
13          // Assignment to m_value is not protected by lock
14          m_value = value;
15      }
16
```

```
17      void SetValue2(std::int32_t value)
18      {
19          std::lock_guard<std::mutex> guard{m_mtx}; // Compliant: local variable
20          m_value = value;
21      }
22
23  private:
24      mutable std::mutex m_mtx;
25      std::int32_t m_value;
26  };
27
28  void Print(std::string const & fname, std::string const & s)
29  {
30      // Compliant: Not only constructing a temporary object
31      std::ofstream{fname}.write(s.c_str(), s.length());
32  }
```

See also

- C++ Core Guidelines [11]: ES.84: Don't (try to) declare a local variable with no name
- C++ Core Guidelines [11]: CP.44: Remember to name your lock_guards and unique_locks

Rule M6-2-2 (required, implementation, partially automated)
Floating-point expressions shall not be directly or indirectly tested for equality or inequality.

See MISRA C++ 2008 [7]

Rule M6-2-3 (required, implementation, automated)
Before preprocessing, a null statement shall only occur on a line by itself; it may be followed by a comment, provided that the first character following the null statement is a white-space character.

See MISRA C++ 2008 [7]

6.6.3 Compound statement or block

Rule M6-3-1 (required, implementation, automated)
The statement forming the body of a switch, while, do ... while or for statement shall be a compound statement.

See MISRA C++ 2008 [7]

6.6.4 Selection statements

Rule M6-4-1 (required, implementation, automated)
An if (condition) construct shall be followed by a compound statement. The else keyword shall be followed by either a compound statement, or another if statement.

See MISRA C++ 2008 [7]

Rule M6-4-2 (required, implementation, automated)
All if ... else if constructs shall be terminated with an else clause.

See MISRA C++ 2008 [7]

Rule M6-4-3 (required, implementation, automated)
A switch statement shall be a well-formed switch statement.

See MISRA C++ 2008 [7]

Rule M6-4-4 (required, implementation, automated)
A switch-label shall only be used when the most closely-enclosing compound statement is the body of a switch statement.

See MISRA C++ 2008 [7]

Rule M6-4-5 (required, implementation, automated)
An unconditional throw or break statement shall terminate every non-empty switch-clause.

See MISRA C++ 2008 [7]

Rule M6-4-6 (required, implementation, automated)
The final clause of a switch statement shall be the default-clause.

See MISRA C++ 2008 [7]

Rule M6-4-7 (required, implementation, automated)
The condition of a switch statement shall not have bool type.

See MISRA C++ 2008 [7]

Note: "'The condition shall be of integral type, enumeration type, or class type. If of class type, the condition is contextually implicitly converted (Clause 4) to an integral or enumeration type."' [C++14 Language Standard, 6.4.2 The switch statement]

Rule A6-4-1 (required, implementation, automated)
A switch statement shall have at least two case-clauses, distinct from the default label.

Rationale

A switch statement constructed with less than two case-clauses can be expressed as an if statement more naturally.

Note that a switch statement with no case-clauses is redundant.

Example

```
1  // $Id: A6-4-1.cpp 289436 2017-10-04 10:45:23Z michal.szczepankiewicz $
2  #include <cstdint>
3  void F1(std::uint8_t choice) noexcept
4  {
5      switch (choice)
6      {
7          default:
8              break;
9      }  // Non-compliant, the switch statement is redundant
10 }
11 void F2(std::uint8_t choice) noexcept
12 {
13     switch (choice)
14     {
15         case 0:
16             // ...
17             break;
18
19         default:
20             // ...
21             break;
22     }  // Non-compliant, only 1 case-clause
23
24     if (choice == 0)  // Compliant, an equivalent if statement
25     {
26         // ...
27     }
28     else
```

```
29      {
30          // ...
31      }
32
33      // ...
34      switch (choice)
35      {
36          case 0:
37              // ...
38              break;
39
40          case 1:
41              // ...
42              break;
43
44          default:
45              // ...
46              break;
47      }  // Compliant
48  }
```

See also

- MISRA C++ 2008 [7]: Rule 6-4-8 Every switch statement shall have at least one case-clause.
- HIC++ v4.0 [9]: 6.1.4 Ensure that a switch statement has at least two case labels, distinct from the default label.

6.6.5 Iteration statements

Rule A6-5-1 (required, implementation, automated)
A for-loop that loops through all elements of the container and does not use its loop-counter shall not be used.

Rationale

A for-loop that simply loops through all elements of the container and does not use its loop-counter is equivalent to a range-based for statement that reduces the amount of code to maintain correct loop semantics.

Example

```
1  // $Id: A6-5-1.cpp 289436 2017-10-04 10:45:23Z michal.szczepankiewicz $
2  #include <cstdint>
3  #include <iterator>
4  void Fn() noexcept
5  {
6      constexpr std::int8_t arraySize = 7;
```

```
7       std::uint32_t array[arraySize] = {0, 1, 2, 3, 4, 5, 6};
8
9       for (std::int8_t idx = 0; idx < arraySize; ++idx)  // Compliant
10      {
11          array[idx] = idx;
12      }
13
14      for (std::int8_t idx = 0; idx < arraySize / 2;
15           ++idx)  // Compliant - for does not loop though all elements
16      {
17          // ...
18      }
19
20      for (std::uint32_t* iter = std::begin(array); iter != std::end(array);
21           ++iter)  // Non-compliant
22      {
23          // ...
24      }
25
26      for (std::int8_t idx = 0; idx < arraySize; ++idx)  // Non-compliant
27      {
28          // ...
29      }
30
31      for (std::uint32_t value :
32           array)  // Compliant - equivalent to non-compliant loops above
33      {
34          // ...
35      }
36
37      for (std::int8_t idx = 0; idx < arraySize; ++idx)  // Compliant
38      {
39          if ((idx % 2) == 0)
40          {
41              // ...
42          }
43      }
44  }
```

See also

- HIC++ v4.0 [9]: 6.2.1 Implement a loop that only uses element values as a range-based loop.
- C++ Core Guidelines [11]: ES.55: Avoid the need for range checking.
- C++ Core Guidelines [11]: ES.71: Prefer a range-for-statement to a for-statement when there is a choice.

Rule A6-5-2 (required, implementation, automated)
A for loop shall contain a single loop-counter which shall not have floating-point type.

Rationale

A for loop without a loop-counter is simply a while loop. If this is the desired behavior, then a while loop is more appropriate.

Floating types, as they should not be tested for equality/inequality, are not to be used as loop-counters.

Example

```
// $Id: A6-5-2.cpp 289436 2017-10-04 10:45:23Z michal.szczepankiewicz $
#include <cstdint>
namespace
{
constexpr std::int32_t xlimit = 20;
constexpr std::int32_t ylimit = 15;
constexpr float zlimit = 2.5F;
constexpr std::int32_t ilimit = 100;
}
void Fn() noexcept
{
    std::int32_t y = 0;

    for (std::int32_t x = 0; x < xlimit && y < ylimit;
         x++, y++)  // Non-compliant, two loop-counters
    {
        // ...
    }

    for (float z = 0.0F; z != zlimit;
         z += 0.1F)  // Non-compliant, float with !=
    {
        // ...
    }

    for (float z = 0.0F; z < zlimit; z += 0.1F)  // Non-compliant, float with <
    {
        // ...
    }

    for (std::int32_t i = 0; i < ilimit; ++i)  // Compliant
    {
        // ...
    }
}
```

See also

- MISRA C++ 2008 [7]: Rule 6-5-1 A for loop shall contain a single loop-counter which shall not have floating type.
- C++ Core Guidelines [11]: ES.72: Prefer a for-statement to a while-statement when there is an obvious loop variable.

Rule M6-5-2 (required, implementation, automated)
If loop-counter is not modified by $--$ or $++$, then, within condition, the loop-counter shall only be used as an operand to $<=$, $<$, $>$ or $>=$.

See MISRA C++ 2008 [7]

Rule M6-5-3 (required, implementation, automated)
The loop-counter shall not be modified within condition or statement.

See MISRA C++ 2008 [7]

Rule M6-5-4 (required, implementation, automated)
The loop-counter shall be modified by one of: $--$, $++$, $-=n$, or $+=n$; where n remains constant for the duration of the loop.

See MISRA C++ 2008 [7]

Note: “n remains constant for the duration of the loop” means that “n” can be either a literal, a constant or constexpr value.

Rule M6-5-5 (required, implementation, automated)
A loop-control-variable other than the loop-counter shall not be modified within condition or expression.

See MISRA C++ 2008 [7]

Rule M6-5-6 (required, implementation, automated)
A loop-control-variable other than the loop-counter which is modified in statement shall have type bool.

See MISRA C++ 2008 [7]

Rule A6-5-3 (advisory, implementation, automated)
Do statements should not be used.

Rationale

Do-statements are bug-prone, as the termination condition is checked at the end and can be overlooked.

Exception

A do-statement may be used in a function-like macro to ensure that its invocation behaves like an expression statement consisting of a function call (see http://c-faq.com/cpp/multistmt.html).

Note: Rule A16-0-1 forbids function-like macros. This exception is kept in case rule A16-0-1 is disabled in a project.

Example

```
1  // $Id: A6-5-3.cpp 291350 2017-10-17 14:31:34Z jan.babst $
2
3  #include <cstdint>
4
5  // Compliant by exception
6  #define SWAP(a, b)              \
7      do                          \
8      {                           \
9          decltype(a) tmp = (a); \
10         (a) = (b);              \
11         (b) = tmp;              \
12     } while (0)
13
14 // Non-compliant
15 #define SWAP2(a, b)         \
16     decltype(a) tmp = (a); \
17     (a) = (b);              \
18     (b) = tmp;
19
20 int main(void)
21 {
22     uint8_t a = 24;
23     uint8_t b = 12;
24
25     if (a > 12)
26         SWAP(a, b);
27
28     // if (a > 12)
29     //     SWAP2(a, b);
30     // Does not compile, because only the first line is used in the body of the
31     // if-statement. In other cases this may even cause a run-time error.
32     // The expansion contain two semicolons in a row, which may be flagged by
33     // compiler warnings.
```

```
34      // Expands to:
35      // if (a > 12)
36      //     decltype(a) tmp = (a);
37      // (a) = (b);
38      // (b) = tmp;;
39
40      return 0;
41  }
```

See also

- C++ Core Guidelines [11]: ES.75: Avoid do-statements.

Rule A6-5-4 (advisory, implementation, automated)
For-init-statement and expression should not perform actions other than loop-counter initialization and modification.

Rationale

If only a loop-counter is used in the for-init-statement and expression, it increases readability and it is easier to understand and maintain code.

Example

```
1  // $Id: A6-5-4.cpp 305629 2018-01-29 13:29:25Z piotr.serwa $
2  #include <cstdint>
3
4  void Fn() noexcept
5  {
6      for (std::int32_t x = 0, MAX=10; x < MAX; x++)  // compliant with A6-5-2, but
       non-compliant with advisory A6-5-4
7      {
8          // ...
9      }
10 }
```

See also

- JSF December 2005 [8]: AV Rule 198: The initialization expression in a for loop will perform no actions other than to initialize the value of a single for loop parameter.
- JSF December 2005 [8]: AV Rule 199: The increment expression in a for loop will perform no action other than to change a single loop parameter to the next value for the loop.

6.6.6 Jump statements

Rule A6-6-1 (required, implementation, automated)
The goto statement shall not be used.

Rationale

Using goto statement significantly complicates the logic, makes the code difficult to read and maintain, and may lead to incorrect resources releases or memory leaks.

Example

```
// $Id: A6-6-1.cpp 289436 2017-10-04 10:45:23Z michal.szczepankiewicz $
#include <cstdint>
namespace
{
constexpr std::int32_t loopLimit = 100;
}
void F1(std::int32_t n) noexcept
{
    if (n < 0)
    {
        // goto exit;  // Non-compliant - jumping to exit from here crosses ptr
        // pointer initialization, compilation
        // error
    }

    std::int32_t* ptr = new std::int32_t(n);
// ...
exit:
    delete ptr;
}
void F2() noexcept
{
    // ...
    goto error;  // Non-compliant
// ...
error:;  // Error handling and cleanup
}
void F3() noexcept
{
    for (std::int32_t i = 0; i < loopLimit; ++i)
    {
        for (std::int32_t j = 0; j < loopLimit; ++j)
        {
            for (std::int32_t k = 0; k < loopLimit; ++k)
            {
                if ((i == j) && (j == k))
                {
                    // ...
                    goto loop_break;  // Non-compliant
                }
            }
```

```
42          }
43      }
44
45  loop_break:;  // ...
46  }
```

See also

- JSF December 2005 [8]: AV Rule 189 The goto statement shall not be used.
- C++ Core Guidelines [11]: ES.76: Avoid goto.
- C++ Core Guidelines [11]: NR.6: Don't: Place all cleanup actions at the end of a function and goto exit.

Rule M6-6-1 (required, implementation, automated)
Any label referenced by a goto statement shall be declared in the same block, or in a block enclosing the goto statement.

See MISRA C++ 2008 [7]

Rule M6-6-2 (required, implementation, automated)
The goto statement shall jump to a label declared later in the same function body.

See MISRA C++ 2008 [7]

Rule M6-6-3 (required, implementation, automated)
The continue statement shall only be used within a well-formed for loop.

See MISRA C++ 2008 [7]

6.7 Declaration

6.7.1 Specifiers

Rule A7-1-1 (required, implementation, automated)
Constexpr or const specifiers shall be used for immutable data declaration.

Rationale

If data is declared to be const or constexpr then its value can not be changed by mistake. Also, such declaration can offer the compiler optimization opportunities.

Note that the constexpr specifier in an object declaration implies const as well.

Example

```
//% $Id: A7-1-1.cpp 289436 2017-10-04 10:45:23Z michal.szczepankiewicz $
#include <cstdint>
#include <limits>
void Fn()
{
    const std::int16_t x1 = 5;      // Compliant
    constexpr std::int16_t x2 = 5;  // Compliant
    std::int16_t x3 =
        5;  // Non-compliant - x3 is not modified but not declared as
            // constant (const or constexpr)
}
```

See also

- C++ Core Guidelines [11]: ES.25: Declare objects const or constexpr unless you want to modify its value later on.
- C++ Core Guidelines [11]: Con.1: By default, make objects immutable.
- C++ Core Guidelines [11]: Con.4: Use const to define objects with values that do not change after construction.

Rule A7-1-2 (required, implementation, automated)
The constexpr specifier shall be used for values that can be determined at compile time.

Rationale

The constexpr specifier declares that it is possible to evaluate the value of the function or variable at compile time, e.g. integral type overflow/underflow, configuration options or some physical constants. The compile-time evaluation can have no side effects so it is more reliable than const expressions.

Note that the constexpr specifier in an object declaration implies const, and when used in a function declaration it implies inline.

Note also that since 2014 C++ Language Standard constexpr specifier in member function declaration no longer implicitly implies that the member function is const.

Example

```
//% $Id: A7-1-2.cpp 289436 2017-10-04 10:45:23Z michal.szczepankiewicz $
#include <cstdint>
std::int32_t Pow1(std::int32_t number)
{
    return (number * number);
}
```

```
constexpr std::int32_t Pow2(
    std::int32_t number)  // Possible compile-time computing
                          // because of constexpr specifier
{
    return (number * number);
}
void Fn()
{
    constexpr std::int16_t i1 = 20;  // Compliant, evaluated at compile-time
    const std::int16_t i2 = 20;  // Non-compliant, possible run-time evaluation
    std::int32_t twoSquare =
        Pow1(2);  // Non-compliant, possible run-time evaluation
    const std::int32_t threeSquare =
        Pow1(3);  // Non-compliant, possible run-time evaluation
    // static_assert(threeSquare == 9, "pow1(3) did not succeed.");  // Value
    // can not be static_assert-ed
    constexpr std::int32_t fiveSquare =
        Pow2(5);  // Compliant, evaluated at compile time
    static_assert(fiveSquare == 25,
                  "pow2(5) did not succeed.");  // Compliant, constexpr
                                                // evaluated at compile time
    // constexpr std::int32_t int32Max =
    // std::numeric_limits<std::int32_t>::max() + 1;  //
    // Compliant - compilation error due to
    // compile-time evaluation (integer overflow)
}
class A
{
  public:
    static constexpr double pi = 3.14159265;  // Compliant - value of PI can be
                                              // determined in compile time

    // constexpr double e = 2.71828182;  // Non-compliant - constexprs need
    // to be static members, compilation error

    constexpr A() = default;  // Compliant
};
```

See also

- C++ Core Guidelines [11]: Con.5: Use constexpr for values that can be computed at compile time.

Rule M7-1-2 (required, implementation, automated)
A pointer or reference parameter in a function shall be declared as pointer to const or reference to const if the corresponding object is not modified.

See MISRA C++ 2008 [7]

See also

- C++ Core Guidelines [11]: Con.3: By default, pass pointers and references to consts.

Rule A7-1-3 (required, implementation, automated)
CV-qualifiers shall be placed on the right hand side of the type that is a typedef or a using name.

Rationale

If the type is a typedef or a using name, placing const or volatile qualifier on the left hand side may result in confusion over what part of the type the qualification applies to.

Example

```
// $Id: A7-1-3.cpp 289436 2017-10-04 10:45:23Z michal.szczepankiewicz $
#include <cstdint>
using IntPtr = std::int32_t*;
using IntConstPtr = std::int32_t* const;
using ConstIntPtr = const std::int32_t*;
void Fn(const std::uint8_t& input)  // Compliant
{
    std::int32_t value1 = 10;
    std::int32_t value2 = 20;

    const IntPtr ptr1 =
        &value1;  // Non-compliant - deduced type is std::int32_t*
                  // const, not const std::int32_t*

    // ptr1 = &value2;  // Compilation error, ptr1 is read-only variable

    IntPtr const ptr2 =
        &value1;  // Compliant - deduced type is std::int32_t* const

    // ptr2 = &value2;  // Compilation error, ptr2 is read-only variable

    IntConstPtr ptr3 = &value1;  // Compliant - type is std::int32_t* const, no
                                 // additional qualifiers needed

    // ptr3 = &value2;  // Compilation error, ptr3 is read-only variable

    ConstIntPtr ptr4 = &value1;  // Compliant - type is const std::int32_t*

    const ConstIntPtr ptr5 = &value1;  // Non-compliant, type is const
                                       // std::int32_t* const, not const const
                                       // std::int32_t*
    ConstIntPtr const ptr6 =
        &value1;  // Compliant - type is const std::int32_t* const
}
```

See also

- HIC++ v4.0 [9]: 7.1.4 Place CV-qualifiers on the right hand side of the type they apply to

Rule A7-1-4 (required, implementation, automated)
The register keyword shall not be used.

Rationale

This feature was deprecated in the 2011 C++ Language Standard [2] and may be withdrawn in a later version.

Moreover, most compilers ignore register specifier and perform their own register assignments.

Example

```
// $Id: A7-1-4.cpp 289448 2017-10-04 11:11:03Z michal.szczepankiewicz $
#include <cstdint>
std::int32_t F1(register std::int16_t number) noexcept  // Non-compliant
{
    return ((number * number) + number);
}
void F2(std::int16_t number) noexcept  // Compliant
{
    register std::int8_t x = 10;       // Non-compliant
    std::int32_t result = F1(number);  // Compliant
    // ...
}
```

See also

- JSF December 2005 [8]: AV Rule 140 The register storage class specifier shall not be used.
- HIC++ v4.0 [9]: 1.3.2 Do not use the register keyword

Rule A7-1-5 (required, implementation, automated)
The auto specifier shall not be used apart from following cases: (1) to declare that a variable has the same type as return type of a function call, (2) to declare that a variable has the same type as initializer of non-fundamental type, (3) to declare parameters of a generic lambda expression, (4) to declare a function template using trailing return type syntax.

Rationale

Using the auto specifier may lead to unexpected type deduction results, and therefore to developers confusion. In most cases using the auto specifier makes the code less readable.

Note that it is allowed to use the auto specifier in following cases:

1. When declaring a variable that is initialized with a function call or initializer of non-fundamental type. Using the auto specifier for implicit type deduction in such cases will ensure that no unexpected implicit conversions will occur. In such case, explicit type declaration would not aid readability of the code.
2. When declaring a generic lambda expression with auto parameters
3. When declaring a function template using trailing return type syntax

Example

```
1  // $Id: A7-1-5.cpp 289436 2017-10-04 10:45:23Z michal.szczepankiewicz $
2  #include <cstdint>
3  #include <vector>
4
5  class A
6  {
7  };
8  void F1() noexcept
9  {
10     auto x1 = 5;      // Non-compliant - initializer is of fundamental type
11     auto x2 = 0.3F;   // Non-compliant - initializer is of fundamental type
12     auto x3 = {8};    // Non-compliant - initializer is of fundamental type
13
14     std::vector<std::int32_t> v;
15     auto x4 = v.size();  // Compliant with case (1) - x4 is of size_t type that
16                          // is returned from v.size() method
17
18     auto a = A{};  // Compliant with case (2)
19
20     auto lambda1 = []() -> std::uint16_t {
21         return 5U;
22     };  // Compliant with case (2) - lambda1 is of non-fundamental lambda
23     // expression type
24     auto x5 = lambda1();  // Compliant with case (1) - x5 is of
25                           // std::uint16_t type
26 }
27 void F2() noexcept
28 {
29     auto lambda1 = [](auto x, auto y) -> decltype(x + y) {
30         return (x + y);
31     };                             // Compliant with cases (2) and (3)
32     auto y1 = lambda1(5.0, 10);  // Compliant with case (1)
33 }
34 template <typename T, typename U>
35 auto F3(T t, U u) noexcept -> decltype(t + u)  // Compliant with case (4)
36 {
37     return (t + u);
38 }
39 template <typename T>
40 class B
41 {
```

```
42    public:
43      T Fn(T t);
44  };
45  template <typename T>
46  auto B<T>::Fn(T t) -> T  // Compliant with case (4)
47  {
48      // ...
49      return t;
50  }
```

See also

- HIC++ v4.0 [9]: 7.1.8 Use auto id = expr when declaring a variable to have the same type as its initializer function call.
- C++ Core Guidelines [11]: Use auto.
- Google C++ Style Guide [12]: Use auto to avoid type names that are noisy, obvious, or unimportant.

Rule A7-1-6 (required, implementation, automated)
The typedef specifier shall not be used.

Rationale

The typedef specifier can not be easily used for defining alias templates. Also, the typedef syntax makes the code less readable.

For defining aliases, as well as template aliases, it is recommended to use the using syntax instead of the typedef.

Note that active issues related to the using syntax are listed below, in the “See also” section.

Example

```
1  // $Id: A7-1-6.cpp 271687 2017-03-23 08:57:35Z piotr.tanski $
2  #include <cstdint>
3  #include <type_traits>
4
5  typedef std::int32_t (*fPointer1)(std::int32_t);  // Non-compliant
6
7  using fPointer2 = std::int32_t (*)(std::int32_t);  // Compliant
8
9  // template<typename T>
10 // typedef std::int32_t (*fPointer3)(T);  // Non-compliant - compilation error
11
12 template <typename T>
13 using fPointer3 = std::int32_t (*)(T);  // Compliant
```

See also

- C++ Core Guidelines [11]: T.43: Prefer using over typedef for defining aliases
- C++ Standard Core Language Active Issues, Revision 96 [18]: 1554. Access and alias templates.
- C++ Standard Core Language Defect Reports and Accepted Issues, Revision 96 [18]: 1558. Unused arguments in alias template specializations.

Rule A7-1-7 (required, implementation, automated)
Each expression statement and identifier declaration shall be placed on a separate line.

Rationale

Declaring an identifier on a separate line makes the identifier declaration easier to find and the source code more readable. Also, combining objects, references and pointers declarations with assignments and function calls on the same line may become confusing.

Exception

It is permitted to declare identifiers in initialization statement of a for loop.

Example

```
// $Id: A7-1-7.cpp 292454 2017-10-23 13:14:23Z michal.szczepankiewicz $
#include <cstdint>
#include <vector>

typedef std::int32_t* ptr;              // Compliant
typedef std::int32_t *pointer, value;  // Non-compliant

void Fn1() noexcept
{
    std::int32_t x = 0;                 // Compliant
    std::int32_t y = 7, *p1 = nullptr;  // Non-compliant
    std::int32_t const *p2, z = 1;      // Non-compliant
}

void Fn2()
{
    std::vector<std::int32_t> v{1, 2, 3, 4, 5};
    for (auto iter{v.begin()}, end{v.end()}; iter != end;
         ++iter)  // Compliant by exception
    {
        // ...
    }
}

void Fn3() noexcept
{
```

```
27      std::int32_t x{5};
28      std::int32_t y{15};  // Non-compliant
29      x++;
30      ++y;  // Non-compliant
31      for (std::int32_t i{0}; i < 100; ++i)
32      {
33          Fn2();  // Compliant
34      }
35  }
```

See also

- HIC++ v4.0 [9]: 7.1.1 Declare each identifier on a separate line in a separate declaration.
- JSF December 2005 [8]: AV Rule 42 Each expression-statement will be on a separate line.
- JSF December 2005 [8]: AV Rule 152: Multiple variable declarations shall not be allowed on the same line.
- C++ Core Guidelines [11]: NL.20: Don't place two statements on the same line.

Rule A7-1-8 (required, implementation, automated)
A non-type specifier shall be placed before a type specifier in a declaration.

Rationale

Placing a non-type specifier, i.e. typedef, friend, constexpr, register, static, extern, thread_local, mutable, inline, virtual, explicit, before type specifiers makes the source code more readable.

Example

```
1  // $Id: A7-1-8.cpp 289436 2017-10-04 10:45:23Z michal.szczepankiewicz $
2  #include <cstdint>
3
4  typedef std::int32_t int1;  // Compliant
5  std::int32_t typedef int2;  // Non-compliant
6
7  class C
8  {
9    public:
10     virtual inline void F1();  // Compliant
11     inline virtual void F2();  // Compliant
12     void virtual inline F3();  // Non-compliant
13   private:
14     std::int32_t mutable x;  // Non-compliant
15     mutable std::int32_t y;  // Compliant
16 };
```

See also

- HIC++ v4.0 [9]: 7.1.3 Do not place type specifiers before non-type specifiers in a declaration.

Rule A7-1-9 (required, implementation, automated)
A class, structure, or enumeration shall not be declared in the definition of its type.

Rationale

Combining a type definition with a declaration of another entity can lead to readability problems and can be confusing for a developer.

Example

```
1  // $Id: A7-1-9.cpp 305629 2018-01-29 13:29:25Z piotr.serwa $
2  #include <cstdint>
3
4  enum class DIRECTION
5  {
6      UP,
7      DOWN
8  } dir;  //non-compliant
9
10 class Foo
11 {
12 public:
13     enum class ONE {AA, BB}; //compliant
14
15     static constexpr enum class TWO {CC, DD} sVar = TWO::CC; // non-compliant
16     static constexpr ONE sVar2 = ONE::AA; //compliant
17 };
18
19
20 struct Bar
21 {
22     std::uint32_t a;
23 } barObj; //non-compliant
24
25 struct Bar2
26 {
27     std::uint32_t a;
28 } bar2Obj, *bar2Ptr; //non-compliant, also with A7-1-7
29
30 struct Foo2
31 {
32     std::uint32_t f;
33 };
34
35 Foo2 foo2Obj; //compliant
```

See also

- JSF December 2005 [8]: AV Rule 141: A class, structure, or enumeration will not be declared in the definition of its type.
- C++ Core Guidelines [11]: C.7: Don't define a class or enum and declare a variable of its type in the same statement.

6.7.2 Enumeration declaration

Rule A7-2-1 (required, implementation, automated)
An expression with enum underlying type shall only have values corresponding to the enumerators of the enumeration.

Rationale

It is unspecified behavior if the evaluation of an expression with enum underlying type yields a value which does not correspond to one of the enumerators of the enumeration.

Additionally, other rules in this standard assume that objects of enum type only contain values corresponding to the enumerators. This rule ensures the validity of these assumptions.

One way of ensuring compliance when converting to an enumeration is to use a switch statement.

Example

```
1  // $Id: A7-2-1.cpp 289436 2017-10-04 10:45:23Z michal.szczepankiewicz $
2  #include <cstdint>
3  enum class E : std::uint8_t
4  {
5      Ok = 0,
6      Repeat,
7      Error
8  };
9  E Convert1(std::uint8_t number) noexcept
10 {
11     E result = E::Ok;  // Compliant
12     switch (number)
13     {
14         case 0:
15         {
16             result = E::Ok;  // Compliant
17             break;
18         }
19         case 1:
20         {
21             result = E::Repeat;  // Compliant
```

```
22              break;
23          }
24          case 2:
25          {
26              result = E::Error;  // Compliant
27              break;
28          }
29          case 3:
30          {
31              constexpr std::int8_t val = 3;
32              result = static_cast<E>(val);  // Non-compliant - value 3 does not
33                                             // correspond to any of E's
34                                             // enumerators
35              break;
36          }
37          default:
38          {
39              result =
40                  static_cast<E>(0);  // Compliant - value 0 corresponds to E::Ok
41              break;
42          }
43      }
44      return result;
45  }
46  E Convert2(std::uint8_t userInput) noexcept
47  {
48      E result = static_cast<E>(userInput);  // Non-compliant - the range of
49                                             // userInput may not correspond to
50                                             // any of E's enumerators
51      return result;
52  }
53  E Convert3(std::uint8_t userInput) noexcept
54  {
55      E result = E::Error;
56      if (userInput < 3)
57      {
58          result = static_cast<E>(userInput);  // Compliant - the range of
59                                               // userInput checked before casting
60                                               // it to E enumerator
61      }
62      return result;
63  }
```

See also

- MISRA C++ 2008 [7]: Rule 7-2-1 An expression with enum underlying type shall only have values corresponding to the enumerators of the enumeration.

Rule A7-2-2 (required, implementation, automated)
Enumeration underlying base type shall be explicitly defined.

Rationale

The enumeration underlying type is implementation-defined, with the only restriction that the type must be able to represent the enumeration values. Although scoped enum will implicitly define an underlying type of int, the underlying base type of enumeration should always be explicitly defined with a type that will be large enough to store all enumerators.

Example

```
// $Id: A7-2-2.cpp 271715 2017-03-23 10:13:51Z piotr.tanski $
#include <cstdint>
enum class E1  // Non-compliant
{
    E10,
    E11,
    E12
};
enum class E2 : std::uint8_t  // Compliant
{
    E20,
    E21,
    E22
};
enum E3  // Non-compliant
{
    E30,
    E31,
    E32
};
enum E4 : std::uint8_t  // Compliant - violating another rule
{
    E40,
    E41,
    E42
};
enum class E5 : std::uint8_t  // Non-compliant - will not compile
{
    E50 = 255,
    // E5_1,  // E5_1 = 256 which is outside of range of underlying type
    // std::uint8_t
    // - compilation error
    // E5_2  // E5_2 = 257 which is outside of range of underlying type
    // std::uint8_t
    // - compilation error
};
```

See also

- HIC++ v4.0 [9]: 7.2.1 Use an explicit enumeration base and ensure that it is large enough to store all enumerators

Rule A7-2-3 (required, implementation, automated)
Enumerations shall be declared as scoped enum classes.

Rationale

If unscoped enumeration enum is declared in a global scope, then its values can redeclare constants declared with the same identifier in the global scope. This may lead to developer's confusion.

Using enum-class as enumeration encloses its enumerators in its inner scope and prevent redeclaring identifiers from outer scope.

Note that enum class enumerators disallow implicit conversion to numeric values.

Example

```
1  // $Id: A7-2-3.cpp 289436 2017-10-04 10:45:23Z michal.szczepankiewicz $
2  #include <cstdint>
3
4  enum E1 : std::int32_t  // Non-compliant
5  {
6      E10,
7      E11,
8      E12
9  };
10
11 enum class E2 : std::int32_t  // Compliant
12 {
13     E20,
14     E21,
15     E22
16 };
17
18 // static std::int32_t E1_0 = 5;  // E1_0 symbol redeclaration, compilation
19 // error
20
21 static std::int32_t e20 = 5;  // No redeclarations, no compilation error
22
23 extern void F1(std::int32_t number)
24 {
25 }
26
27 void F2()
28 {
29     F1(0);
30
31     F1(E11);  // Implicit conversion from enum to std::int32_t type
32
```

```
33      // f1(E2::E2_1);  // Implicit conversion not possible, compilation error
34
35      F1(static_cast<std::int32_t>(
36          E2::E21));  // Only explicit conversion allows to
37                      // pass E2_1 value to f1() function
38  }
```

See also

- C++ Core Guidelines [11]: Enum.3: Prefer class enums over "'plain"' enums.

Rule A7-2-4 (required, implementation, automated)
In an enumeration, either (1) none, (2) the first or (3) all enumerators shall be initialized.

Rationale

Explicit initialization of only some enumerators in an enumeration, and relying on compiler to initialize the remaining ones, may lead to developer's confusion.

Example

```
1  //% $Id: A7-2-4.cpp 271715 2017-03-23 10:13:51Z piotr.tanski $
2  #include <cstdint>
3  enum class Enum1 : std::uint32_t
4  {
5      One,
6      Two = 2,  // Non-compliant
7      Three
8  };
9  enum class Enum2 : std::uint32_t  // Compliant (none)
10 {
11     One,
12     Two,
13     Three
14 };
15 enum class Enum3 : std::uint32_t  // Compliant (the first)
16 {
17     One = 1,
18     Two,
19     Three
20 };
21 enum class Enum4 : std::uint32_t  // Compliant (all)
22 {
23     One = 1,
24     Two = 2,
25     Three = 3
26 };
```

See also

- MISRA C++ 2008 [7]: Rule 8-5-3 In an enumerator list, the = construct shall not be used to explicitly initialize members other than the first, unless all items are explicitly initialized.
- HIC++ v4.0 [9]: 7.2.2 Initialize none, the first only or all enumerators in an enumeration.

Rule A7-2-5 (advisory, design, non-automated)
Enumerations should be used to represent sets of related named constants.

Rationale

Explicit declaration of constants as an enumeration clearly shows that they are related, which enhances readability and maintenance.

Note: Using switch statement on an enumeration is a common case and such an approach helps to detect errors, see M6-4-6.

Example

```
1  //% $Id: A7-2-5.cpp 305629 2018-01-29 13:29:25Z piotr.serwa $
2  #include <cstdint>
3  //compliant
4  enum class WebpageColors: std::uint32_t
5  {
6      Red,
7      Blue,
8      Green
9  };
10 //non-compliant
11 enum class Misc: std::uint32_t
12 {
13     Yellow,
14     Monday,
15     Holiday
16 };
```

See also

- JSF December 2005 [8]: AV Rule 148: Enumeration types shall be used instead of integer types (and constants) to select from a limited series of choices.
- C++ Core Guidelines [11]: Enum.2: Use enumerations to represent sets of related named constants.

6.7.3 Namespaces

Rule M7-3-1 (required, implementation, automated)
The global namespace shall only contain main, namespace declarations and extern "C" declarations.

See MISRA C++ 2008 [7]

Rule M7-3-2 (required, implementation, automated)
The identifier main shall not be used for a function other than the global function main.

See MISRA C++ 2008 [7]

Rule M7-3-3 (required, implementation, automated)
There shall be no unnamed namespaces in header files.

See MISRA C++ 2008 [7]

Rule M7-3-4 (required, implementation, automated)
Using-directives shall not be used.

See MISRA C++ 2008 [7]

See: Using-directive [16] concerns an inclusion of specific namespace with all its types, e.g. using namespace std.

Rule A7-3-1 (required, implementation, automated)
All overloads of a function shall be visible from where it is called.

Rationale

Additional identifiers introduced by a using declaration makes only prior declarations of this identifier visible. Any potential subsequent declarations will not be added to the current scope, which may lead to unexpected results and developers confusion.

Overriding or overloading a member function in a derived class causes other member functions with the same name to be hidden. Thus, a potential function call may result in a different function being called depending on if the call was made using the derived or base class reference/pointer. Introducing hidden names into the derived class by a using declaration helps to avoid such misleading situations.

Example

```
// $Id: A7-3-1.cpp 312801 2018-03-21 16:17:05Z michal.szczepankiewicz $
#include <cstdint>

class Base
{
public:
    void P(uint32_t);

    virtual void V(uint32_t);
    virtual void V(double);
};

class NonCompliant : public Base
{
public:
    //hides P(uint32_t) when calling from the
    //derived class
    void P(double);
    //hides V(uint32_t) when calling from the
    //derived class
    void V(double) override;
};

class Compliant : public Base
{
public:
    //both P(uint32_t) and P(double) available
    //from the derived class
    using Base::P;
    void P(double);

    //both P(uint32_t) and P(double)
    using Base::V;
    void V(double) override;
};

void F1()
{
    NonCompliant d{};
    d.P(0U); // D::P (double) called
    Base& b{d};
    b.P(0U); // NonCompliant::P (uint32_t) called

    d.V(0U); // D::V (double) called
    b.V(0U); // NonCompliant::V (uint32_t) called
}

void F2()
{
    Compliant d{};
    d.P(0U); // Compliant::P (uint32_t) called
```

```
52      Base& b{d};
53      b.P(0U); // Compliant::P (uint32_t) called
54
55      d.V(0U); // Compliant::V (uint32_t) called
56      b.V(0U); // Compliant::V (uint32_t) called
57  }
58
59  namespace NS
60  {
61      void F(uint16_t);
62  }
63
64  //includes only preceding declarations into
65  //the current scope
66  using NS::F;
67
68  namespace NS
69  {
70      void F(uint32_t);
71  }
72
73  void B(uint32_t b)
74  {
75      //non-compliant, only F(uint16_t) is available
76      //in this scope
77      F(b);
78  }
```

See also

- MISRA C++ 2008 [7]: 7-3-5: Multiple declarations for an identifier in the same namespace shall not straddle a using-declaration for that identifier.
- HIC++ v4.0 [9]: 13.1.1: Ensure that all overloads of a function are visible from where it is called.

Rule M7-3-6 (required, implementation, automated)
Using-directives and using-declarations (excluding class scope or function scope using-declarations) shall not be used in header files.

See MISRA C++ 2008 [7]

See: Using-declaration [16] concerns an inclusion of specific type, e.g. using std::string.

6.7.4 The asm declaration

Rule A7-4-1 (required, implementation, automated)
The asm declaration shall not be used.

Rationale

Inline assembly code restricts the portability of the code.

Example

```
// $Id: A7-4-1.cpp 289436 2017-10-04 10:45:23Z michal.szczepankiewicz $
#include <cstdint>
std::int32_t Fn1(std::int32_t b) noexcept
{
    std::int32_t ret = 0;
    // ...
    asm("pushq %%rax \n"
        "movl %0, %%eax \n"
        "addl %1, %%eax \n"
        "movl %%eax, %0 \n"
        "popq %%rax"
        : "=r"(ret)
        : "r"(b));  // Non-compliant
    return ret;
}
std::int32_t Fn2(std::int32_t b) noexcept
{
    std::int32_t ret = 0;
    // ...
    ret += b;  // Compliant - equivalent to asm(...) above
    return ret;
}
```

See also

- HIC++ v4.0 [9]: 7.5.1 Do not use the asm declaration.

Rule M7-4-1 (required, implementation, non-automated)
All usage of assembler shall be documented.

See MISRA C++ 2008 [7]

Rule M7-4-2 (required, implementation, automated)
Assembler instructions shall only be introduced using the asm declaration.

See MISRA C++ 2008 [7]

Rule M7-4-3 (required, implementation, automated)
Assembly language shall be encapsulated and isolated.

See MISRA C++ 2008 [7]

6.7.5 Linkage specification

Rule M7-5-1 (required, implementation, non-automated)
A function shall not return a reference or a pointer to an automatic variable (including parameters), defined within the function.

See MISRA C++ 2008 [7]

Rule M7-5-2 (required, implementation, non-automated)
The address of an object with automatic storage shall not be assigned to another object that may persist after the first object has ceased to exist.

See MISRA C++ 2008 [7]

Note: C++ specifies that binding a temporary object (e.g. automatic variable returned from a function) to a reference to const prolongs the lifetime of the temporary to the lifetime of the reference.

Note: Rule 7-5-2 concerns C++11 smart pointers, i.e. std::unique_ptr, std::shared_ptr and std::weak_ptr, too.

See also

- C++ Core Guidelines [11]: F.45: Don't return a T&&.

Rule A7-5-1 (required, implementation, automated)
A function shall not return a reference or a pointer to a parameter that is passed by reference to const.

Rationale

"[...] Where a parameter is of const reference type a temporary object is introduced if needed (7.1.6, 2.13, 2.13.5, 8.3.4, 12.2)." [C++14 Language Standard [3]]

Any attempt to dereferencing an object which outlived its scope will lead to undefined behavior.

References to const bind to both lvalues and rvalues, so functions that accept parameters passed by reference to const should expect temporary objects too.

Returning a pointer or a reference to such an object leads to undefined behavior on accessing it.

Example

```
// $Id: A7-5-1.cpp 289436 2017-10-04 10:45:23Z michal.szczepankiewicz $
#include <cstdint>
class A
{
  public:
    explicit A(std::uint8_t n) : number(n) {}
    ~A() { number = 0U; }
    // Implementation

  private:
    std::uint8_t number;
};
const A& Fn1(const A& ref) noexcept  // Non-compliant - the function returns a
                                     // reference to const reference parameter
                                     // which may bind to temporary objects.
                                     // According to C++14 Language Standard, it
// is undefined whether a temporary object is introduced for const
// reference
// parameter
{
    // ...
    return ref;
}
const A& Fn2(A& ref) noexcept  // Compliant - non-const reference parameter does
                               // not bind to temporary objects, it is allowed
                               // that the function returns a reference to such
                               // a parameter
{
    // ...
    return ref;
}
const A* Fn3(const A& ref) noexcept  // Non-compliant - the function returns a
// pointer to const reference parameter
// which may bind to temporary objects.
// According to C++14 Language Standard, it
// is undefined whether a temporary object is introduced for const
// reference
// parameter
{
    // ...
    return &ref;
}
template <typename T>
T& Fn4(T& v)  // Compliant - the function will not bind to temporary objects
{
    // ...
    return v;
```

```
48 }
49 void F() noexcept
50 {
51     A a{5};
52     const A& ref1 = Fn1(a);  // fn1 called with an lvalue parameter from an
53                              // outer scope, ref1 refers to valid object
54     const A& ref2 = Fn2(a);  // fn2 called with an lvalue parameter from an
55                              // outer scope, ref2 refers to valid object
56     const A* ptr1 = Fn3(a);  // fn3 called with an lvalue parameter from an
57                              // outer scope, ptr1 refers to valid object
58     const A& ref3 = Fn4(a);  // fn4 called with T = A, an lvalue parameter from
59                              // an outer scope, ref3 refers to valid object
60
61     const A& ref4 = Fn1(A{10});  // fn1 called with an rvalue parameter
62                                  // (temporary), ref3 refers to destroyed object
63                                  // A const& ref5 = fn2(A{10});  // Compilation
64                                  // error - invalid initialization of non-const
65                                  // reference
66     const A* ptr2 = Fn3(A{15});  // fn3 called with an rvalue parameter
67                                  // (temporary), ptr2 refers to destroyted
68                                  // object
69     // const A& ref6 = fn4(A{20});  // Compilation error - invalid
70     // initialization of non-const reference
71 }
```

See also

- MISRA C++ 2008 [7]: A function shall not return a reference or a pointer to a parameter that is passed by reference or const reference.

Rule A7-5-2 (required, implementation, automated)
Functions shall not call themselves, either directly or indirectly.

Rationale

As the stack space is limited resource, use of recursion may lead to stack overflow at run-time. It also may limit the scalability and portability of the program.

Recursion can be replaced with loops, iterative algorithms or worklists.

Exception

Recursion in variadic template functions used to process template arguments does not violate this rule, as variadic template arguments are evaluated at compile time and the call depth is known.

Recursion of a constexpr function does not violate this rule, as it is evaluated at compile time.

Example

```
1  // $Id: A7-5-2.cpp 289436 2017-10-04 10:45:23Z michal.szczepankiewicz $
2  #include <cstdint>
3  static std::int32_t Fn1(std::int32_t number);
4  static std::int32_t Fn2(std::int32_t number);
5  static std::int32_t Fn3(std::int32_t number);
6  static std::int32_t Fn4(std::int32_t number);
7  std::int32_t Fn1(std::int32_t number)
8  {
9      if (number > 1)
10     {
11         number = number * Fn1(number - 1);  // Non-compliant
12     }
13
14     return number;
15 }
16 std::int32_t Fn2(std::int32_t number)
17 {
18     for (std::int32_t n = number; n > 1; --n)  // Compliant
19     {
20         number = number * (n - 1);
21     }
22
23     return number;
24 }
25 std::int32_t Fn3(std::int32_t number)
26 {
27     if (number > 1)
28     {
29         number = number * Fn3(number - 1);  // Non-compliant
30     }
31
32     return number;
33 }
34 std::int32_t Fn4(std::int32_t number)
35 {
36     if (number == 1)
37     {
38         number = number * Fn3(number - 1);  // Non-compliant
39     }
40
41     return number;
42 }
43 template <typename T>
44 T Fn5(T value)
45 {
46     return value;
47 }
48 template <typename T, typename... Args>
49 T Fn5(T first, Args... args)
50 {
51     return first + Fn5(args...);  // Compliant by exception - all of the
```

```
                                    // arguments are known during compile time
}
std::int32_t Fn6() noexcept
{
    std::int32_t sum = Fn5<std::int32_t, std::uint8_t, float, double>(
        10, 5, 2.5, 3.5);  // An example call to variadic template function
    // ...
    return sum;
}
constexpr std::int32_t Fn7(std::int32_t x, std::int8_t n)
{
    if (n >= 0)
    {
        x += x;
        return Fn5(x, --n);  // Compliant by exception - recursion evaluated at
                             // compile time
    }
    return x;
}
```

See also

- MISRA C++ 2008 [7]: Rule 7-5-4 Functions should not call themselves, either directly or indirectly.
- JSF December 2005 [8]: AV Rule 119 Functions shall not call themselves, either directly or indirectly (i.e. recursion shall not be allowed).
- HIC++ v4.0 [9]: 5.2.2 Ensure that functions do not call themselves, either directly or indirectly.

6.7.6 Attributes

Rule A7-6-1 (required, implementation, automated)
Functions declared with the [[noreturn]] attribute shall not return.

Rationale

The C++ standard specifies that functions with the [[noreturn]] attribute shall not return. Returning from such a function can be prohibited in the following way: throwing an exception, entering an infinite loop, or calling another function with the [[noreturn]] attribute. Returning from such a function leads to undefined behavior.

```
// $Id: A7-6-1.cpp 305629 2018-01-29 13:29:25Z piotr.serwa $
#include <cstdint>
#include <exception>

class PositiveInputException : public std::exception {};

```

```
[[noreturn]] void f(int i)  //non-compliant
{
    if (i > 0)
    {
        throw PositiveInputException();
    }
    //undefined behaviour for non-positive i
}

[[noreturn]] void g(int i) //compliant
{
    if (i > 0)
    {
      throw "Received positive input";
    }

    while(1)
    {
        //do processing
    }
}
```

See also

- SEI CERT C++ Coding Standard [10]: MSC53-CPP: Do not return from a function declared [[noreturn]].

6.8 Declarators

6.8.0 General

Rule M8-0-1 (required, implementation, automated)
An init-declarator-list or a member-declarator-list shall consist of a single init-declarator or member-declarator respectively.

See MISRA C++ 2008 [7]

6.8.2 Ambiguity resolution

Rule A8-2-1 (required, implementation, automated)
When declaring function templates, the trailing return type syntax shall be used if the return type depends on the type of parameters.

Rationale

Use of trailing return type syntax avoids a fully qualified return type of a function along with the typename keyword.

Example

```
// $Id: A8-2-1.cpp 289436 2017-10-04 10:45:23Z michal.szczepankiewicz $
#include <cstdint>
template <typename T>
class A
{
  public:
    using Type = std::int32_t;

    Type F(T const&) noexcept;
    Type G(T const&) noexcept;
};
template <typename T>
typename A<T>::Type A<T>::F(T const&) noexcept  // Non-compliant
{
    // Implementation
}
template <typename T>
auto A<T>::G(T const&) noexcept -> Type  // Compliant
{
    // Implementation
}
```

See also

- HIC++ v4.0 [9]: 7.1.7 Use a trailing return type in preference to type disambiguation using typename.

6.8.3 Meaning of declarators

Rule M8-3-1 (required, implementation, automated)
Parameters in an overriding virtual function shall either use the same default arguments as the function they override, or else shall not specify any default arguments.

See MISRA C++ 2008 [7]

Note: Overriding non-virtual functions in a subclass is called function “hiding” or “redefining”. It is prohibited by A10-2-1.

6.8.4 Function definitions

Rule A8-4-1 (required, implementation, automated)
Functions shall not be defined using the ellipsis notation.

Rationale

Passing arguments via an ellipsis bypasses the type checking performed by the compiler. Additionally, passing an argument with non-POD class type leads to undefined behavior.

Variadic templates offer a type-safe alternative for ellipsis notation. If use of a variadic template is not possible, function overloading or function call chaining can be considered.

Example

```
1  // $Id: A8-4-1.cpp 289436 2017-10-04 10:45:23Z michal.szczepankiewicz $
2
3  void Print1(char* format, ...)  // Non-compliant - variadic arguments are used
4  {
5      // ...
6  }
7
8  template <typename First, typename... Rest>
9  void Print2(const First& first, const Rest&... args)  // Compliant
10 {
11     // ...
12 }
```

See also

- MISRA C++ 2008 [7]: Rule 8-4-1 Functions shall not be defined using the ellipsis notation.
- HIC++ v4.0 [9]: 14.1.1 Use variadic templates rather than an ellipsis.
- C++ Core Guidelines [11]: Type.8: Avoid reading from varargs or passing vararg arguments. Prefer variadic template parameters instead.
- C++ Core Guidelines [11]: F.55: Don't use va_arg arguments.

Rule M8-4-2 (required, implementation, automated)
The identifiers used for the parameters in a re-declaration of a function shall be identical to those in the declaration.

See MISRA C++ 2008 [7]

Rule A8-4-2 (required, implementation, automated)
All exit paths from a function with non-void return type shall have an explicit return statement with an expression.

Rationale

In a function with non-void return type, return expression gives the value that the function returns. The absence of a return with an expression leads to undefined behavior (and the compiler may not give an error).

Exception

A function may additionally exit due to exception handling (i.e. a throw statement).

Example

```
// $Id: A8-4-2.cpp 289436 2017-10-04 10:45:23Z michal.szczepankiewicz $
#include <cstdint>
#include <stdexcept>
std::int32_t F1() noexcept  // Non-compliant
{
}
std::int32_t F2(std::int32_t x) noexcept(false)
{
    if (x > 100)
    {
        throw std::logic_error("Logic Error");  // Compliant by exception
    }

    return x;  // Compliant
}
std::int32_t F3(std::int32_t x, std::int32_t y)
{
    if (x > 100 || y > 100)
    {
        throw std::logic_error("Logic Error");  // Compliant by exception
    }
    if (y > x)
    {
        return (y - x);  // Compliant
    }
    return (x - y);  // Compliant
}
```

See also

- MISRA C++ 2008 [7]: Rule 8-4-3 All exit paths from a function with non-void return type shall have an explicit return statement with an expression.
- SEI CERT C++ [10]: MSC52-CPP. Value-returning functions must return a value from all exit paths.

Rule M8-4-4 (required, implementation, automated)
A function identifier shall either be used to call the function or it shall be preceded by &.

See MISRA C++ 2008 [7]

Rule A8-4-3 (advisory, design, non-automated)
Common ways of passing parameters should be used.

Rationale

Using common and well-understood parameter passing patterns as summarised in the following table helps meeting developer expectations.

	cheap to copy or move only	cheap to move	expensive to move
in	f(X)	f(const X &)	
in/out	f(X &)		
out	X f()		f(X &)
consume	f(X &&)		
forward	template<typename T> f(T &&)		

Parameter passing

Example

```
1  // $Id: A8-4-3.cpp 308906 2018-02-23 15:34:15Z christof.meerwald $
2
3  #include <algorithm>
4  #include <array>
5  #include <cstdint>
6  #include <numeric>
7  #include <string>
8  #include <vector>
9
10 // Compliant: passing cheap-to-copy parameter by value
11 int32_t Increment(int32_t i)
12 {
13     return i + 1;
14 }
15
16 // Compliant: passing expensive to copy parameter by reference to const
17 int32_t Sum(const std::vector<int32_t> &v)
18 {
19     return std::accumulate(v.begin(), v.end(), 0);
20 }
21
22 // Compliant: passing in-out parameter by reference
23 void Decrement(int32_t &i)
24 {
```

```
25      --i;
26  }
27
28  // Compliant: returning out parameter by value
29  std::string GetGreeting()
30  {
31      return "Hello";
32  }
33
34  struct A
35  {
36      std::string text;
37      std::array<std::string, 1000> arr;
38  };
39
40  // Expensive to move "out" parameter passed by reference. If
41  // intentional, violation of A8-4-8 needs to be explained
42  void InitArray(std::array<std::string, 1000> &arr,
43                 const std::string &text)
44  {
45      std::for_each(arr.begin(), arr.end(), [&text] (std::string &s) {
46              s = text;
47          });
48  }
49
50  // Compliant: passing in-out parameter by reference
51  void PopulateA(A &a)
52  {
53      InitArray(a.arr, a.text);
54  }
```

See also

- C++ Core Guidelines [11]: F.16: Prefer simple and conventional ways of passing information

Rule A8-4-4 (advisory, design, automated)
Multiple output values from a function should be returned as a struct or tuple.

Rationale

Returning multiple values from a function using a struct or tuple clearly states output parameters and allows to avoid confusion of passing them as a reference in a function call. Returning a struct or tuple will not have an additional overhead for compilers that support return-value-optimization.

In C++14, a returned tuple can be conveniently processed using `std::tie` at the call site, which will put the tuple elements directly into existing local variables. In C++17,

structured bindings allow to initialize local variables directly from members or elements of a returned struct or tuple.

Note: For return types representing an abstraction, a struct should be preferred over a generic tuple.

Note: This rule applies equally to `std::pair`, which is a special kind of tuple for exactly two elements.

Example

```
// $Id: A8-4-4.cpp 289816 2017-10-06 11:19:42Z michal.szczepankiewicz $

#include <tuple>

// Non-compliant, remainder returned as the output parameter
int Divide1(int dividend, int divisor, int& remainder)
{
    remainder = dividend % divisor;
    return dividend / divisor;
}

// Compliant, both quotient and remainder returned as a tuple
std::tuple<int, int> Divide2(int dividend, int divisor)
{
    return std::make_tuple(dividend / divisor, dividend % divisor);
}

// Compliant since C++17, return tuple using list-initialization
// std::tuple<int, int> Divide3(int dividend, int divisor)
//{
//    return { dividend / divisor, dividend % divisor };
//}

int main()
{
    int quotient, remainder;
    std::tie(quotient, remainder) = Divide2(26, 5);  // store in local variables
    // auto [quotient, remainder] = Divide3(26, 5); // since C++17, by
    // structured bindings
    return 0;
}
```

See also

- C++ Core Guidelines [11]: F.21: To return multiple ”out” values, prefer returning a tuple or struct.

Rule A8-4-5 (required, design, automated)
”consume” parameters declared as X && shall always be moved from.

Rationale

A ”consume” parameter is declared with a type of rvalue reference to non-const non-template type (X &&). This documents that the value will be consumed in the function (i.e. left in a moved-from state) and requires an explicit ’std::move’ at the call site if an lvalue is passed to the function (an rvalue reference can implicitly bind only to an rvalue).

Note: Other operations may be performed on the ”consume” parameter before being moved.

Example

```
1  // $Id: A8-4-5.cpp 305588 2018-01-29 11:07:35Z michal.szczepankiewicz $
2
3  #include <string>
4  #include <vector>
5
6  class A
7  {
8    public:
9      explicit A(std::vector<std::string> &&v)
10       : m_v{std::move(v)} // Compliant, move from consume parameter
11     {
12     }
13
14   private:
15     std::vector<std::string> m_v;
16 };
17
18 class B
19 {
20   public:
21     explicit B(std::vector<std::string> &&v)
22       : m_v{v} // Non-Compliant, consume parameter not moved from
23     {
24     }
25
26     std::vector<std::string> m_v;
27 };
```

See also

- C++ Core Guidelines [11]: F.18: For ”consume” parameters, pass by X&& and std::move the parameter

Rule A8-4-6 (required, design, automated)
”forward” parameters declared as T && shall always be forwarded.

Rationale

A ”forward” parameter is declared with a type of forwarding reference (i.e. an rvalue reference to non-const template type (T &&)). As a forwarding reference can bind to both lvalues and rvalues, preserving lvalue-ness and cv qualifications, it is useful when forwarding a value to another function using ”std::forward”.

However, as the parameter can bind to anything, it should only be used for forwarding without performing any other operations on the parameter.

Note: A forwarding parameter can also be declared via ”auto &&” in a generic lambda

Example

```
// $Id: A8-4-6.cpp 305588 2018-01-29 11:07:35Z michal.szczepankiewicz $

#include <string>
#include <vector>

class A
{
  public:
    explicit A(std::vector<std::string> &&v);
};

class B
{
  public:
    explicit B(const std::vector<std::string> &v);
};


template<typename T, typename ... Args>
T make(Args && ... args)
{
    return T{std::forward<Args>(args) ...}; // Compliant, forwarding args
}

int main()
{
    make<A>(std::vector<std::string>{ });

    std::vector<std::string> v;
    make<B>(v);
}
```

See also

- C++ Core Guidelines [11]: F.19: For ”forward” parameters, pass by TP&& and only std::forward the parameter
- A18-9-2 in section 6.18.9

Rule A8-4-7 (required, design, automated)
"in" parameters for "cheap to copy" types shall be passed by value.

Rationale

Passing an argument by value documents that the argument won't be modified. Copying the value (instead of passing by reference to const) also ensures that no indirection is needed in the function body to access the value.

For the purpose of this rule, "cheap to copy" is defined as a trivially copyable type that is no longer than two words (i.e. pointers).

Example

```
1  // $Id: A8-4-7.cpp 305588 2018-01-29 11:07:35Z michal.szczepankiewicz $
2
3  #include <cstdint>
4  #include <iostream>
5  #include <string>
6
7  // Compliant, pass by value
8  void output(std::uint32_t i)
9  {
10     std::cout << i << '\n';
11 }
12
13 // Non-Compliant, std::string is not trivially copyable
14 void output(std::string s)
15 {
16     std::cout << s << '\n';
17 }
18
19 struct A
20 {
21     std::uint32_t v1;
22     std::uint32_t v2;
23 };
24
25 // Non-Compliant, A is trivially copyable and no longer than two words
26 void output(const A &a)
27 {
28     std::cout << a.v1 << ", " << a.v2 << '\n';
29 }
```

See also

- C++ Core Guidelines [11]: F.16: For "in" parameters, pass cheaply-copied types by value and others by reference to const
- JSF December 2005 [8]: AV Rule 116: Small, concrete-type arguments (two or three words in size) should be passed by value if changes made to formal parameters should not be reflected in the calling function.

- JSF December 2005 [8]: AV Rule 117.1: An object should be passed as const T& if the function should not change the value of the object.
- A18-9-2 in section 6.18.9

Rule A8-4-8 (required, design, automated)
Output parameters shall not be used.

Rationale

Output parameters are passed to a function as non-const references or pointers that can denote either in-out or out-only parameter. Using return value prevents from possible misuse of such a parameter.

Note: Prefer returning non-value types (i.e. types in a inheritance hierarchy) as std::shared_ptr or std::unique_ptr.

Example

```
// $Id: A8-4-8.cpp 306164 2018-02-01 15:04:53Z christof.meerwald $

#include <iostream>
#include <vector>
// Compliant, return value
std::vector<int> SortOutOfPlace(const std::vector<int>& inVec);

// Non-compliant: return value as an out-parameter
void FindAll(const std::vector<int>& inVec, std::vector<int>& outVec);

struct B
{
};

struct BB
{
    B GetB() const& { return obj; }
    B&& GetB() && { return std::move(obj); }

    B obj;
};

// Non-compliant: returns a dangling reference
BB&& MakeBb1()
{
    return std::move(BB());
}

// Compliant: uses compiler copy-ellision
BB MakeBb2()
{
    return BB();
```

```
}

int main()
{
    BB x = MakeBb2();

    auto cpd = x.GetB();          // copied value
    auto mvd = MakeBb2().GetB();  // moved value

    return 0;
}
```

See also

- C++ Core Guidelines [11]: F.20: For ”out” output values, prefer return values to output parameters.

Rule A8-4-9 (required, design, automated)
”in-out” parameters declared as T & shall be modified.

Rationale

An ”in-out” parameter is declared with a type of reference to non-const. This means that a fully constructed object is passed into the function that can be read as well as modified.

Note: Completely replacing the passed in object without reading any data from it would make it an ”out” parameter instead and is not considered compliant with this rule, also see rule: A8-4-8

Example

```
// $Id: A8-4-9.cpp 306178 2018-02-01 15:52:25Z christof.meerwald $

#include <cstdint>
#include <numeric>
#include <string>
#include <vector>

// Non-Compliant: does not modify the "in-out" parameter
int32_t Sum(std::vector<int32_t> &v)
{
    return std::accumulate(v.begin(), v.end(), 0);
}

// Compliant: Modifying "in-out" parameter
void AppendNewline(std::string &s)
{
    s += '\n';
}
```

```
19
20 // Non-Compliant: Replacing parameter value
21 void GetFileExtension(std::string &ext)
22 {
23     ext = ".cpp";
24 }
```

See also

- C++ Core Guidelines [11]: F.17: For ”in-out” parameters, pass by reference to non-const
- JSF December 2005 [8]: AV Rule 117.2: An object should be passed as T& if the function may change the value of the object.

Rule A8-4-10 (required, design, automated)
A parameter shall be passed by reference if it can’t be NULL

Rationale

Passing a parameter by pointer suggests that it can be NULL. If it can’t be NULL (i.e. it’s not optional) it should therefore be passed by reference instead. Only parameters that can be NULL shall be passed by pointer.

Note: The C++ Library Fundamentals TS v2 defines `std::observer_ptr` as a near drop-in replacement for raw pointers that makes it explicit that the object is not owned by the pointer.

Note: `boost::optional` supports reference types, and in C++17 `std::optional` can be used in conjunction with `std::reference_wrapper` (using `std::optional` with a value type would create an undesirable copy of the object)

Example

```
1  // $Id: A8-4-10.cpp 307966 2018-02-16 16:03:46Z christof.meerwald $
2
3  #include <cstdint>
4  #include <numeric>
5  #include <vector>
6
7  // Non-Compliant: non-optional parameter passed by pointer
8  int32_t Sum(const std::vector<int32_t> *v)
9  {
10     return std::accumulate(v->begin(), v->end(), 0);
11 }
12
13 // Compliant: non-optional parameter passed by reference
14 int32_t Sum(const std::vector<int32_t> &v)
15 {
16     return std::accumulate(v.begin(), v.end(), 0);
17 }
```

See also

- C++ Core Guidelines [11]: F.60: Prefer T* over T& when "no argument" is a valid option
- JSF December 2005 [8]: AV Rule 118: Arguments should be passed via pointers if NULL values are possible.
- JSF December 2005 [8]: AV Rule 118.1: An object should be passed as const T* if its value should not be modified.
- JSF December 2005 [8]: AV Rule 118.2: An object should be passed as T* if its value may be modified.

Rule A8-4-11 (required, design, automated)
A smart pointer shall only be used as a parameter type if it expresses lifetime semantics

Rationale

If the object passed into the function is merely used without affecting the lifetime, it is preferable to pass it by reference or raw pointer instead.

Keeping a copy of a `std::shared_ptr` or moving a `std::unique_ptr` would be examples that affect the lifetime.

Note: When an object whose lifetime is managed by a non-local smart pointer is passed by reference or raw pointer, care needs to be taken that the lifetime of the object doesn't end during the duration of the called function. In the case of a `std::shared_ptr` this can be achieved by keeping a local copy of the `shared_ptr`.

Exception

A non-owning smart pointer, like `std::observer_ptr` from the C++ Library Fundamentals TS v2, that documents the non-owning property of the parameter does not violate this rule.

Example

```
1  // $Id: A8-4-11.cpp 307966 2018-02-16 16:03:46Z christof.meerwald $
2
3  #include <cstdint>
4  #include <memory>
5  #include <numeric>
6  #include <vector>
7
8  class A
9  {
10   public:
11     void do_stuff();
12 };
```

```
13
14  // Non-Compliant: passing object as smart pointer
15  void foo(std::shared_ptr<A> a)
16  {
17      if (a)
18      {
19          a->do_stuff();
20      }
21      else
22      {
23          // ...
24      }
25  }
26
27  // Compliant: passing as raw pointer instead
28  void bar(A *a)
29  {
30      if (a != nullptr)
31      {
32          a->do_stuff();
33      }
34      else
35      {
36          // ...
37      }
38  }
39
40
41  class B
42  {
43    public:
44      void add_a(std::shared_ptr<A> a)
45      {
46        m_v.push_back(a);
47      }
48
49    private:
50      std::vector<std::shared_ptr<A>> m_v;
51  };
52
53  // Compliant: storing the shared pointer (affecting lifetime)
54  void bar(B &b, std::shared_ptr<A> a)
55  {
56      b.add_a(a);
57  }
```

See also

- C++ Core Guidelines [11]: R.30: Take smart pointers as parameters only to explicitly express lifetime semantics.

- C++ Core Guidelines [11]: R.37: Do not pass a pointer or reference obtained from an aliased smart pointer.
- C++ Core Guidelines [11]: F.7: For general use, take T* or T& arguments rather than smart pointers.
- A18-5-2 in section 6.18.5

Rule A8-4-12 (required, design, automated)
A std::unique_ptr shall be passed to a function as: (1) a copy to express the function assumes ownership (2) an lvalue reference to express that the function replaces the managed object.

Rationale

Transferring ownership in the (1) case is unconditional. A temporary `std::unique_ptr` is constructed implicitly and move-initialized from the caller's `std::unique_ptr` and then passed to the function. This guarantees that the caller's `std::unique_ptr` object is empty.

Passing an lvalue reference is suggested to be used if a called function is supposed to replace the object managed by the passed `std::unique_ptr`, e.g. call assignment operator or `reset` method. Otherwise, it is recommended to pass an lvalue reference to the underlying object instead, see A8-4-11, A8-4-10.

Note: Passing a const lvalue reference to `std::unique_ptr` does not take ownership and does not allow to replace the managed object. Also, the const qualifier does not apply to the underlying object, but to the smart pointer itself. It is suggested to pass a const lvalue reference to the underlying object instead, see A8-4-11, A8-4-10.

Exception

It is allowed to transfer ownership by passing a `std::unique_ptr` by an rvalue reference in case this reference is moved into a `std::unique_ptr` object inside the called function.

Example

```
1  // $Id: A8-4-12.cpp 308795 2018-02-23 09:27:03Z michal.szczepankiewicz $
2
3  #include <memory>
4  #include <iostream>
5
6  //compliant, transfers an ownership
7  void Value(std::unique_ptr<int> v) { }
8
9  //compliant, replaces the managed object
10 void Lv1(std::unique_ptr<int>& v)
11 {
12     v.reset();
13 }
14
```

```
15 //non-compliant, does not replace the managed object
16 void Lv2(std::unique_ptr<int>& v) {}
17
18
19 //compliant by exception
20 void Rv1(std::unique_ptr<int>&& r)
21 {
22     std::unique_ptr<int> v(std::move(r));
23 }
24
25 //non-compliant
26 void Rv2(std::unique_ptr<int>&& r) {}
27
28 int main(void)
29 {
30     auto sp = std::make_unique<int>(7);
31     Value(std::move(sp));
32     //sp is empty
33
34     auto sp2 = std::make_unique<int>(9);
35     Rv1(std::move(sp2));
36     //sp2 is empty, because it was moved from in Rv1 function
37
38     auto sp3 = std::make_unique<int>(9);
39     Rv2(std::move(sp3));
40     //sp3 is not empty, because it was not moved from in Rv1 function
41
42     return 0;
43 }
```

See also

- HIC++ v4.0 [9]: 8.2.4: Do not pass std::unique_ptr by const reference.
- C++ Core Guidelines [11]: R.32: Take a unique_ptr<widget> parameter to express that a function assumes ownership of a widget.
- C++ Core Guidelines [11]: R.33: Take a unique_ptr<widget>& parameter to express that a function reseats the widget.
- C++ Core Guidelines [11]: I.11: Never transfer ownership by a raw pointer (T*) or reference (T&).

Rule A8-4-13 (required, design, automated)
A std::shared_ptr shall be passed to a function as: (1) a copy to express the function shares ownership (2) an lvalue reference to express that the function replaces the managed object (3) a const lvalue reference to express that the function retains a reference count.

Rationale

Passing a `std::shared_ptr` by value (1) is clear and makes ownership sharing explicit.

Passing an lvalue reference (2) to `std::shared_ptr` is suggested to be used if a called function replaces the managed object on at least one code path, e.g. call assignment operator or `reset` method. Otherwise, it is recommended to pass an lvalue reference to the underlying object instead, see A8-4-11, A8-4-10.

Functions that take a const lvalue reference (3) to `std::shared_ptr` as a parameter are supposed to copy it to another `std::shared_ptr` on at least one code path, otherwise the parameter should be passed by a const lvalue reference to the underlying object instead, see A8-4-11, A8-4-10.

Example

```
// $Id: A8-4-13.cpp 308795 2018-02-23 09:27:03Z michal.szczepankiewicz $

#include <memory>
#include <iostream>

//compliant, explicit ownership sharing
void Value(std::shared_ptr<int> v) { }

//compliant, replaces the managed object
void Lv1(std::shared_ptr<int>& v)
{
    v.reset();
}

//non-compliant, does not replace the managed object
//shall be passed by int& so that API that does not
//extend lifetime of an object is not polluted
//with smart pointers
void Lv2(std::shared_ptr<int>& v)
{
    ++(*v);
}

//compliant, shared_ptr copied in the called function
void Clv1(const std::shared_ptr<int>& v)
{
    Value(v);
}

//non-compliant, const lvalue reference not copied
//to a shared_ptr object on any code path
//shall be passed by const int&
void Clv2(const std::shared_ptr<int>& v)
{
    std::cout << *v << std::endl;
}
```

```
37
38 //non-compliant
39 void Rv1(std::shared_ptr<int>&& r) {}
```

See also

- C++ Core Guidelines [11]: R.34: Take a shared_ptr<widget> parameter to express that a function is part owner.
- C++ Core Guidelines [11]: R.35: Take a shared_ptr<widget>& parameter to express that a function might reseat the shared pointer.
- C++ Core Guidelines [11]: R.36: Take a const shared_ptr<widget>& parameter to express that it might retain a reference count to the object.

Rule A8-4-14 (required, design, non-automated)
Interfaces shall be precisely and strongly typed.

Rationale

Using precise and strong types in interfaces helps using them correctly.

A large number of parameters of fundamental type (particularly of arithmetic type) can be an indication of bad interface design as it does not make it obvious what the units are, and there is no way for the compiler to warn when parameters are passed in the wrong order (as the types are the same or implicitly convertible).

When several parameters are related, combining the parameters into a separate user-defined type should be considered.

Similarly, a type of pointer to `void` does not provide any type safety and alternatives like a pointer to a common base class or using a (potentially constrained) template parameter should be considered.

Example

```
1  // $Id: A8-4-14.cpp 326058 2018-07-16 07:52:31Z christof.meerwald $
2
3  #include <cstdint>
4  #include <chrono>
5
6  // Non-compliant: unit of duration not obvious
7  void Sleep(std::uint32_t duration);
8
9  // Compliant: strongly typed
10 void Sleep(std::chrono::seconds duration);
11
12
13 // Non-compliant: list of related parameters with same type
14 void SetAlarm(std::uint32_t year, std::uint32_t month, std::uint32_t day,
15               std::uint32_t hour, std::uint32_t minute, std::uint32_t second);
```

```
16
17 // Compliant: strongly typed
18 void SetAlarm(std::chrono::system_clock::time_point const & when);
```

See also

- C++ Core Guidelines [11]: I.4: Make interfaces precisely and strongly typed

6.8.5 Initializers

Rule A8-5-0 (required, implementation, automated)
All memory shall be initialized before it is read.

Rationale

Objects with automatic or dynamic storage duration are default-initialized if no initializer is specified. Default initialization produces indeterminate values for objects of neither class nor array types. Default initialization of array types leads to default initialization of each array element. Reading from indeterminate values may produce undefined behavior.

Thus, all local variables, member variables, or objects allocated dynamically must be explicitly initialized before their values are read, unless they are of class type or array of non-fundamental type. It is recommended practice to initialize all such objects immediately when they are defined.

Note: Zero-initialization will happen before any other initialization for any objects with static or thread-local storage duration. Thus, such objects need not be explicitly initialized.

Example

```
1  // $Id: A8-5-0.cpp 307536 2018-02-14 12:35:11Z jan.babst $
2  #include <cstdint>
3  #include <string>
4
5  static std::int32_t zero;  // Compliant - Variable with static storage duration
6                             // is zero-initialized.
7
8  void local()
9  {
10     std::int32_t a;     // No initialization
11     std::int32_t b{};   // Compliant - zero initialization
12
13     b = a;      // Non-compliant - uninitialized memory read
14     a = zero;   // Compliant - a is zero now
15     b = a;      // Compliant - read from initialized memory
16
```

```
17      std::string s;  // Compliant - default constructor is a called
18      // read from s
19  }
20
21  void dynamic()
22  {
23      // Note: These examples violate A18-5-2
24
25      auto const a = new std::int32_t;    // No initialization
26      auto const b = new std::int32_t{};  // Compliant - zero initialization
27
28      *b = *a;    // Non-compliant - uninitialized memory read
29      *a = zero;  // Compliant - a is zero now
30      *b = *a;    // Compliant - read from initialized memory
31
32      delete b;
33      delete a;
34
35      auto const s =
36          new std::string;  // Compliant - default constructor is a called
37      // read from *s
38      delete s;
39  }
40
41  // Members of Bad are default-initialized by the (implicitly generated) default
42  // constructor. Note that this violates A12-1-1.
43  struct Bad
44  {
45      std::int32_t a;
46      std::int32_t b;
47  };
48
49  // Compliant - Members of Good are explicitly initialized.
50  // This also complies to A12-1-1.
51  struct Good
52  {
53      std::int32_t a{0};
54      std::int32_t b{0};
55  };
56
57  void members()
58  {
59      Bad bad;  // Default constructor is called, but members a not initialized
60
61      bad.b = bad.a;  // Non-compliant - uninitialized memory read
62      bad.a = zero;   // Compliant - bad.a is zero now
63      bad.b = bad.a;  // Compliant - read from initialized memory
64
65      Good good;  // Default constructor is called and initializes members
66
67      std::int32_t x = good.a;  // Compliant
```

```
68      std::int32_t y = good.b;  // Compliant
69  }
```

See also

- MISRA C++ 2008 [7]: 8-5-1: All variables shall have a defined value before they are used.
- HIC++ v4.0 [9]: 8.4.1: Do not access an invalid object or an object with indeterminate value
- JSF December 2005 [8]: AV Rule 142: All variables shall be initialized before use.
- SEI CERT C++ Coding Standard [10]: EXP53-CPP: Do not read uninitialized memory
- C++ Core Guidelines [11]: ES.20: Always initialize an object
- ISO/IEC 14882:2014 [3]: 8.5: [dcl.init]

Rule A8-5-1 (required, implementation, automated)
In an initialization list, the order of initialization shall be following: (1) virtual base classes in depth and left to right order of the inheritance graph, (2) direct base classes in left to right order of inheritance list, (3) non-static data members in the order they were declared in the class definition.

Rationale

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

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.

Note that “The order of derivation is relevant only to determine the order of default initialization by constructors and cleanup by destructors.” [C++14 Language Standard [3]]

Example

```
1  // $Id: A8-5-1.cpp 271696 2017-03-23 09:23:09Z piotr.tanski $
2  #include <cstdint>
3  #include <string>
4  class A
5  {
```

```
};
class B
{
};
class C : public virtual B, public A
{
  public:
    C() : B(), A(), s() {}  // Compliant

    // C() : A(), B() { }  // Non-compliant - incorrect order of initialization

  private:
    std::string s;
};
class D
{
};
class E
{
};
class F : public virtual A, public B, public virtual D, public E
{
  public:
    F() : A(), D(), B(), E(), number1(0), number2(0U) {}  // Compliant
    F(F const& oth)
            : B(), E(), A(), D(), number1(oth.number1), number2(oth.number2)
    {
    }  // Non-compliant - incorrect
       // order of initialization

  private:
    std::int32_t number1;
    std::uint8_t number2;
};
```

See also

- HIC++ v4.0 [9]:12.4.4 Write members in an initialization list in the order in which they are declared

Rule M8-5-2 (required, implementation, automated)
Braces shall be used to indicate and match the structure in the non-zero initialization of arrays and structures.

See MISRA C++ 2008 [7]

Rule A8-5-2 (required, implementation, automated)
Braced-initialization {}, without equals sign, shall be used for variable initialization.

Rationale

Braced-initialization using {} braces is simpler and less ambiguous than other forms of initialization. It is also safer, because it does not allow narrowing conversions for numeric values, and it is immune to C++'s most vexing parse.

The use of an equals sign for initialization misleads into thinking that an assignment is taking place, even though it is not. For built-in types like int, the difference is academic, but for user-defined types, it is important to explicitly distinguish initialization from assignment, because different function calls are involved.

Note that most vexing parse is a form of syntactic ambiguity resolution in C++, e.g. "Class c()" could be interpreted either as a variable definition of class "Class" or a function declaration which returns an object of type "Class".

Note that in order to avoid grammar ambiguities, it is highly recommended to use only braced-initialization {} within templates.

Exception

If a class declares both a constructor taking std::initializer_list argument and a constructor which invocation will be ignored in favor of std::initializer_list constructor, this rule is not violated by calling a constructor using () parentheses, see A8-5-4.

Example

```
// $Id: A8-5-2.cpp 289436 2017-10-04 10:45:23Z michal.szczepankiewicz $
#include <cstdint>
#include <initializer_list>
void F1() noexcept
{
    std::int32_t x1 =
        7.9;  // Non-compliant - x1 becomes 7 without compilation error
    // std::int32_t y {7.9};  // Compliant - compilation error, narrowing
    std::int8_t x2{50};     // Compliant
    std::int8_t x3 = {50};  // Non-compliant - std::int8_t x3 {50} is equivalent
                            // and more readable
    std::int8_t x4 =
        1.0;  // Non-compliant - implicit conversion from double to std::int8_t
    std::int8_t x5 = 300;  // Non-compliant - narrowing occurs implicitly
    std::int8_t x6(x5);    // Non-compliant
}
class A
{
  public:
    A(std::int32_t first, std::int32_t second) : x{first}, y{second} {}

```

```
  private:
    std::int32_t x;
    std::int32_t y;
};
struct B
{
    std::int16_t x;
    std::int16_t y;
};
class C
{
  public:
    C(std::int32_t first, std::int32_t second) : x{first}, y{second} {}
    C(std::initializer_list<std::int32_t> list) : x{0}, y{0} {}

  private:
    std::int32_t x;
    std::int32_t y;
};
void F2() noexcept
{
    A a1{1, 5};     // Compliant - calls constructor of class A
    A a2 = {1, 5};  // Non-compliant - calls a default constructor of class A
                    // and not copy constructor or assignment operator.
    A a3(1, 5);     // Non-compliant
    B b1{5, 0};     // Compliant - struct members initialization
    C c1{2, 2};     // Compliant - C(std::initializer_list<std::int32_t>)
                    // constructor is
                    // called
    C c2(2, 2);     // Compliant by exception - this is the only way to call
                    // C(std::int32_t, std::int32_t) constructor
    C c3{{}};  // Compliant - C(std::initializer_list<std::int32_t>) constructor
               // is
               // called with an empty initializer_list
    C c4({2, 2});  // Compliant by exception -
                   // C(std::initializer_list<std::int32_t>)
                   // constructor is called
};
template <typename T, typename U>
void F1(T t, U u) noexcept(false)
{
    std::int32_t x = 0;
    T v1(x);  // Non-compliant
    T v2{x};  // Compliant - v2 is a variable
    // auto y = T(u);  // Non-compliant - is it construction or cast?
    // Compilation error
};
void F3() noexcept
{
    F1(0, "abcd");  // Compile-time error, cast from const char* to int
}
```

See also

- C++ Core Guidelines [11]: ES.23 Prefer the {} initializer syntax.
- C++ Core Guidelines [11]: T.68: Use {} rather than () within templates to avoid ambiguities.
- C++ Core Guidelines [11]: ES.64: Use the T{e} notation for construction.
- Effective Modern C++ [13]: Item 7. Distinguish between () and {} when creating objects.

Rule A8-5-3 (required, implementation, automated)
A variable of type auto shall not be initialized using {} or ={} braced-initialization.

Rationale

If an initializer of a variable of type auto is enclosed in braces, then the result of type deduction may lead to developer confusion, as the variable initialized using {} or ={} will always be of std::initializer_list type.

Note that some compilers, e.g. GCC or Clang, can implement this differently - initializing a variable of type auto using {} will deduce an integer type, and initializing using ={} will deduce a std::initializer_list type. This is desirable type deduction which will be introduced into the C++ Language Standard with C++17.

Example

```
1  // $Id: A8-5-3.cpp 289436 2017-10-04 10:45:23Z michal.szczepankiewicz $
2  #include <cstdint>
3  #include <initializer_list>
4  void Fn() noexcept
5  {
6      auto x1(10);   // Compliant - the auto-declared variable is of type int, but
7                     // not compliant with A8-5-2.
8      auto x2{10};   // Non-compliant - according to C++14 standard the
9                     // auto-declared variable is of type std::initializer_list.
10                    // However, it can behave differently on different compilers.
11     auto x3 = 10;  // Compliant - the auto-declared variable is of type int, but
12                    // non-compliant with A8-5-2.
13     auto x4 = {10};  // Non-compliant - the auto-declared variable is of type
14                      // std::initializer_list, non-compliant with A8-5-2.
15     std::int8_t x5{10};  // Compliant
16 }
```

See also

- Effective Modern C++ [13]: Item 2. Understand auto type deduction.
- Effective Modern C++ [13]: Item 7. Distinguish between () and {} when creating objects.

Rule A8-5-4 (advisory, implementation, automated)
If a class has a user-declared constructor that takes a parameter of type std::initializer_list, then it shall be the only constructor apart from special member function constructors.

Rationale

If an object is initialized using {} braced-initialization, the compiler strongly prefers constructor taking parameter of type `std::initializer_list` to other constructors. Thus, if it is defined in the class, it is initially a sole member of the candidate set of the two-phase overload resolution. Only if no viable `std::initializer_list` is found, the rest of constructors are considered in the second overload resolution.

Such a case can be non-intuitive for developers and can lead to reviewers' confusion on which constructor was intended to be called.

If other constructors (besides the `std::initializer_list` one and special member functions) are declared in a class, then it is suggested to use, e.g. the `std::vector<int>( {1,1} )` syntax instead of `std::vector<int> v{1, 1}`, which makes the intent clear.

Example

```
// $Id: A8-5-4.cpp 319328 2018-05-15 10:30:25Z michal.szczepankiewicz $
#include <cstdint>
#include <initializer_list>
#include <vector>

#include <iostream>

//non-compliant, there are other constructors
//apart from initializer_list one defined
class A
{
public:
    A() = default;
    A(std::size_t num1, std::size_t num2) : x{num1}, y{num2} {}
    A(std::initializer_list<std::size_t> list) : x{list.size()}, y{list.size()} {
     }
private:
    std::size_t x;
    std::size_t y;
};

class B
{
public:
    B() = default;
    B(std::initializer_list<std::size_t> list) : collection{list} { }

private:
    std::vector<std::size_t> collection;
```

```
};

void F1() noexcept
{
    A a1{};      // Calls A::A()
    A a2{{}};    // Calls A::A(std::initializer_list<std::size_t>)
    A a3{0, 1};  // Calls A::A(std::initializer_list<std::size_t>), not
    recommended
    A a4({0, 1});// Calls A::A(std::initializer_list<std::size_t>), recommended
    A a5(0, 1);  // Calls A::A(std::size_t, std::size_t), compliant with A8-5-2
    by exception
}
void F2() noexcept
{
    B b1{};       // Calls B::B()
    B b2{{}};     // Calls B::B(std::initializer_list<std::size_t>)
    B b3{1, 2};   // Calls B::B(std::initializer_list<std::size_t>), not
    recommended
    B b4({1, 2});   // Calls B::B(std::initializer_list<std::size_t>),
    recommended
}
```

See also

- Effective Modern C++ [13]: Item 7. Distinguish between () and {} when creating objects.
- ISO/IEC 14882:2014 [3]: 13.3.1.7: [over.match.list]

6.9 Classes

6.9.3 Member function

Rule M9-3-1 (required, implementation, automated)
Const member functions shall not return non-const pointers or references to class-data.

See MISRA C++ 2008 [7]

Note: This rule applies to smart pointers, too.

Note: “The class-data for a class is all non-static member data and any resources acquired in the constructor or released in the destructor.” [MISRA C++ 2008 [7]]

Rule A9-3-1 (required, implementation, partially automated)
Member functions shall not return non-const "raw" pointers or references to private or protected data owned by the class.

Rationale

By implementing class interfaces with member functions the implementation retains more control over how the object state can be modified and helps to allow a class to be maintained without affecting clients. Returning a handle to data that is owned by the class allows for clients to modify the state of the object without using an interface.

Note that this rule applies to data that are owned by the class (i.e. are class-data). Non-const handles to objects that are shared between different classes may be returned.

See: Ownership.

Exception

Classes that mimic smart pointers and containers do not violate this rule.

Example

```
1  // $Id: A9-3-1.cpp 289436 2017-10-04 10:45:23Z michal.szczepankiewicz $
2  #include <cstdint>
3  #include <memory>
4  #include <utility>
5  class A
6  {
7    public:
8      explicit A(std::int32_t number) : x(number) {}
9      // Implementation
10     std::int32_t&
11     GetX() noexcept  // Non-compliant - x is a resource owned by the A class
12     {
13         return x;
14     }
15
16   private:
17     std::int32_t x;
18 };
19 void Fn1() noexcept
20 {
21     A a{10};
22     std::int32_t& number = a.GetX();
23     number = 15;  // External modification of private class data
24 }
25 class B
26 {
27   public:
28     explicit B(std::shared_ptr<std::int32_t> ptr) : sharedptr(std::move(ptr)) {}
29     // Implementation
30     std::shared_ptr<std::int32_t> GetSharedPtr() const
```

```
31          noexcept  // Compliant - sharedptr is a variable being shared between
32                    // instances
33      {
34          return sharedptr;
35      }
36
37    private:
38      std::shared_ptr<std::int32_t> sharedptr;
39  };
40  void Fn2() noexcept
41  {
42      std::shared_ptr<std::int32_t> ptr = std::make_shared<std::int32_t>(10);
43      B b1{ptr};
44      B b2{ptr};
45      *ptr = 50;  // External modification of ptr which shared between b1 and b2
46                  // instances
47      auto shared = b1.GetSharedPtr();
48      *shared = 100;  // External modification of ptr which shared between b1 and
49                      // b2 instances
50  }
51  class C
52  {
53    public:
54      explicit C(std::int32_t number)
55              : ownedptr{std::make_unique<std::int32_t>(number)}
56      {
57      }
58      // Implementation
59      const std::unique_ptr<std::int32_t>& GetOwnedPtr() const
60          noexcept  // Non-compliant - only unique_ptr is const, the object that
61                    // it is pointing to is modifiable
62      {
63          return ownedptr;
64      }
65      const std::int32_t& GetData() const noexcept  // Compliant
66      {
67          return *ownedptr;
68      }
69
70    private:
71      std::unique_ptr<std::int32_t> ownedptr;
72  };
73  void Fn3() noexcept
74  {
75      C c{10};
76      const std::int32_t& data = c.GetData();
77      // data = 20;  // Can not modify data, it is a const reference
78      const std::unique_ptr<std::int32_t>& ptr = c.GetOwnedPtr();
79      *ptr = 20;  // Internal data of class C modified
80  }
```

See also

- MISRA C++ 2008 [7]: Rule 9-3-2 Member functions shall not return non-const handles to class-data.
- JSF December 2005 [8]: AV Rule 112: Function return values should not obscure resource ownership.

Rule M9-3-3 (required, implementation, automated)
If a member function can be made static then it shall be made static, otherwise if it can be made const then it shall be made const.

See MISRA C++ 2008 [7]

Note: Static methods can only modify static members of a class, they are not able to access data of a class instance.

Note: Const methods can only modify static members of a class or mutable-declared members of a class instance.

See also

- C++ Core Guidelines [11]: Con.2: By default, make member functions const.

6.9.5 Unions

Rule A9-5-1 (required, implementation, automated)
Unions shall not be used.

Rationale

Unions are not type safe and their usage can be misleading and easily misinterpreted by developers.

Exception

It is allowed to use tagged unions until `std::variant` is available in the C++ Standard Library (C++17)

Example

```
1 // $Id: A9-5-1.cpp 305588 2018-01-29 11:07:35Z michal.szczepankiewicz $
2
3 #include <cstdint>
4 // Compliant
5 struct Tagged
6 {
7     enum class TYPE
```

```
8       {
9           UINT,
10          FLOAT
11      };
12      union {
13          uint32_t u;
14          float f;
15      };
16      TYPE which;
17  };
18
19  int main(void)
20  {
21      Tagged un;
22
23      un.u = 12;
24      un.which = Tagged::TYPE::UINT;
25
26      un.u = 3.14f;
27      un.which = Tagged::TYPE::FLOAT;
28
29      return 0;
30  }
```

See also

- MISRA C++ 2008 [7]: M9-5-1: Unions shall not be used
- JSF December 2005 [8]: AV Rule 153: Bit-fields shall have explicitly unsigned integral or enumeration types only
- C++ Core Guidelines [11]: C.181: Avoid “naked” unions
- C++ Core Guidelines [11]: C.182: Use anonymous unions to implement tagged unions
- C++ Core Guidelines [11]: Type.7: Avoid naked union: Use variant instead.

6.9.6 Bit-fields

Rule M9-6-1 (required, implementation, non-automated)
When the absolute positioning of bits representing a bit-field is required, then the behavior and packing of bit-fields shall be documented.

See MISRA C++ 2008 [7]

Rule A9-6-1 (required, design, partially automated)
Data types used for interfacing with hardware or conforming to communication protocols shall be trivial, standard-layout and only contain members of types with defined sizes.

Rationale

When the layout of data types is important, only those types that have a defined size shall be used (see A3-9-1, this excludes `bool`, `wchar_t`, pointers, and pointers to members). Enumeration types may be used if they have been explicitly declared with an underlying type that has a defined size.

Note: As the use of bit-fields is only allowed for interfacing with hardware or conforming to communication protocols, this restriction on types also applies to bit-fields, see A9-6-2.

Note: The signed exact-width integer types like `std::int16_t` are guaranteed to have a two's complement representation.

Example

```
1  // $Id: A9-6-1.cpp 319312 2018-05-15 08:29:17Z christof.meerwald $
2  #include <cstdint>
3
4  enum class E1 : std::uint8_t
5  {
6      E11,
7      E12,
8      E13
9  };
10 enum class E2 : std::int16_t
11 {
12     E21,
13     E22,
14     E23
15 };
16 enum class E3
17 {
18     E31,
19     E32,
20     E33
21 };
22 enum E4
23 {
24     E41,
25     E42,
26     E43
27 };
28
29 class C
30 {
```

```
31    public:
32      std::int32_t a : 2;    // Compliant
33      std::uint8_t b : 2U;   // Compliant
34
35      bool c : 1;  // Non-compliant - the size of bool is implementation defined
36
37      char d : 2;  // Non-compliant
38      wchar_t e : 2;  // Non-compliant - the size of wchar_t is implementation
        defined
39
40      E1 f1 : 2;       // Compliant
41      E2 f2 : 2;       // Compliant
42      E3 f3 : 2;       // Non-compliant - E3 enum class does not explicitly define
43                       // underlying type
44      E4 f4 : 2;  // Non-compliant - E4 enum does not explicitly define underlying
45                  // type
46  };
47
48  struct D
49  {
50      std::int8_t a;              // Compliant
51
52      bool b;                     // Non-compliant - the size of bool is
53                                  // implementation defined
54
55      std::uint16_t c1 : 8;       // Compliant
56      std::uint16_t c2 : 8;       // Compliant
57  };
58
59  void Fn() noexcept
60  {
61      C c;
62      c.f1 = E1::E11;
63  }
```

See also

- MISRA C++ 2008 [7]: A9-6-2: Bit-fields shall be either bool type or an explicitly unsigned or signed integral type
- JSF December 2005 [8]: AV Rule 154: Bit-fields shall have explicitly unsigned integral or enumeration types only
- HIC++ v4.0 [9]: 9.2.1: Declare bit-fields with an explicitly unsigned integral or enumeration type

Rule A9-6-2 (required, design, non-automated)
Bit-fields shall be used only when interfacing to hardware or conforming to communication protocols.

Rationale

Usage of bit-fields increases code complexity and certain aspects of bit-field manipulation can be error prone and implementation-defined. Hence a bit-field usage is reserved only when interfacing to hardware or conformance to communication protocols

Note: A9-6-1 restricts the types allowed to be used in these contexts.

See also

- JSF December 2005 [8]: AV Rule 155: Bit-fields will not be used to pack data into a word for the sole purpose of saving space.

Rule M9-6-4 (required, implementation, automated)
Named bit-fields with signed integer type shall have a length of more than one bit.

See MISRA C++ 2008 [7]

Note: The signed exact-width integer types like `std::int16_t` are guaranteed to have a two's complement representation (see also A9-6-1). In this case, a single bit signed bit-field contains only a sign bit, thus it can represent values either (`-1`) or (`0`). Therefore, to avoid developers' confusion, it is recommended to use unsigned types for single bit bit-fields.

6.10 Derived Classes

6.10.0 General

Rule A10-0-1 (required, design, non-automated)
Public inheritance shall be used to implement "is-a" relationship.

Rationale

Public and non-public inheritance have a very different application and it shall be used accordingly.

See: `Is-a-relationship`, `Has-a-relationship`

See also

- JSF December 2005 [8]: AV Rule 91: Public inheritance will be used to implement "is-a" relationships.

Rule A10-0-2 (required, design, non-automated)
Membership or non-public inheritance shall be used to implement "has-a" relationship.

Rationale

Public and non-public inheritance have a very different application and it shall be used accordingly.

See: Is-a-relationship, Has-a-relationship

See also

- JSF December 2005 [8]: AV Rule 93: "has-a" or "is-implemented-in-terms-of" relationships will be modeled through membership or non-public inheritance.

6.10.1 Multiple base Classes

Rule A10-1-1 (required, implementation, automated)
Class shall not be derived from more than one base class which is not an interface class.

Rationale

Multiple inheritance exposes derived class to multiple implementations. This makes the code more difficult to maintain.

See: Diamond-Problem, Interface-Class

Example

```
1  // $Id: A10-1-1.cpp 289436 2017-10-04 10:45:23Z michal.szczepankiewicz $
2  #include <cstdint>
3  class A
4  {
5    public:
6      void F1() noexcept(false) {}
7
8    private:
9      std::int32_t x{0};
10     std::int32_t y{0};
11 };
12 class B
13 {
14   public:
15     void F2() noexcept(false) {}
16
17   private:
```

```
18      std::int32_t x{0};
19  };
20  class C : public A,
21            public B  // Non-compliant - A and B are both not interface classes
22  {
23  };
24  class D
25  {
26    public:
27      virtual ~D() = 0;
28      virtual void F3() noexcept = 0;
29      virtual void F4() noexcept = 0;
30  };
31  class E
32  {
33    public:
34      static constexpr std::int32_t value{10};
35
36      virtual ~E() = 0;
37      virtual void F5() noexcept = 0;
38  };
39  class F : public A,
40            public B,
41            public D,
42            public E  // Non-compliant - A and B are both not interface classes
43  {
44  };
45  class G : public A,
46            public D,
47            public E  // Compliant - D and E are interface classes
48  {
49  };
```

See also

- JSF December 2005 [8]: AV Rule 88 Multiple inheritance shall only be allowed in the following restricted form: n interfaces plus m private implementations, plus at most one protected implementation.
- HIC++ v4.0 [9]: 10.3.1 Ensure that a derived class has at most one base class which is not an interface class.
- C++ Core Guidelines [11]: C.135: Use multiple inheritance to represent multiple distinct interfaces.

Rule M10-1-1 (advisory, implementation, automated)
Classes should not be derived from virtual bases.

See MISRA C++ 2008 [7]

Rule M10-1-2 (required, implementation, automated)
A base class shall only be declared virtual if it is used in a diamond hierarchy.

See MISRA C++ 2008 [7]

Rule M10-1-3 (required, implementation, automated)
An accessible base class shall not be both virtual and non-virtual in the same hierarchy.

See MISRA C++ 2008 [7]

6.10.2 Member name lookup

Rule M10-2-1 (advisory, implementation, automated)
All accessible entity names within a multiple inheritance hierarchy should be unique.

See MISRA C++ 2008 [7]

Rule A10-2-1 (required, implementation, automated)
Non-virtual public or protected member functions shall not be redefined in derived classes.

Rationale

A non-virtual member function specifies an invariant over the hierarchy. It cannot be overridden in derived classes, but it can be hidden by a derived class member (data or function) with the same identifier. The effect of this hiding is to defeat polymorphism by causing an object to behave differently depending on which interface is used to manipulate it, resulting in unnecessary complexity and error.

Note that a maintenance change to a private implementation detail could impact clients of the base class, and often it will be the case that those clients may not be in a position to fix the problem. Therefore, redefinitions of functions which are private in the base class are not affected by this rule.

Exception

Redefinition of functions from private inheritance do not violate this rule.

Example

```
// $Id: A10-2-1.cpp 317123 2018-04-23 08:48:11Z ilya.burylov $
class A
{
  public:
    virtual ~A() = default;
    void F() noexcept {}
    virtual void G() noexcept {}
  private:
    void H() noexcept {}
};
class B : public A
{
  public:
    void
    F() noexcept {} // Non-compliant - F() function from A class hidden by B
    class
    void G() noexcept override {} // Compliant - G() function from A class
                                  // overridden by B class
  private:
    void H() noexcept {} // Compliant - H() function is private in A class
};
class C : private A
{
  public:
    F() noexcept {} // Compliant by exception - private inheritance
};
void Fn1(A& object) noexcept
{
    object.F();  // Calls F() function from A
    object.G();  // Calls G() function from B
}
void Fn2() noexcept
{
    B b;
    Fn1(b);
}
```

See also

- JSF December 2005 [8]: AV Rule 94 An inherited nonvirtual function shall not be redefined in a derived class.
- C++ Core Guidelines [11]: ES.12: Do not reuse names in nested scopes.

6.10.3 Virtual functions

Rule A10-3-1 (required, implementation, automated)
Virtual function declaration shall contain exactly one of the three specifiers: (1) virtual, (2) override, (3) final.

Rationale

Specifying more than one of these three specifiers along with virtual function declaration is redundant and a potential source of errors.

It is recommended to use the virtual specifier only for new virtual function declaration, the override specifier for overrider declaration, and the final specifier for final overrider declaration.

Note that this applies to virtual destructors and virtual operators, too.

Example

```
// $Id: A10-3-1.cpp 289436 2017-10-04 10:45:23Z michal.szczepankiewicz $
class A
{
  public:
    virtual ~A() {}                          // Compliant
    virtual void F() noexcept = 0;           // Compliant
    virtual void G() noexcept final = 0;     // Non-compliant - virtual final pure
                                             // function is redundant
    virtual void
    H() noexcept final  // Non-compliant - function is virtual and final
    {
    }
    virtual void K() noexcept  // Compliant
    {
    }
    virtual void J() noexcept {}
    virtual void M() noexcept  // Compliant
    {
    }
    virtual void Z() noexcept  // Compliant
    {
    }
    virtual A& operator+=(A const& rhs) noexcept  // Compliant
    {
        // ...
        return *this;
    }
};
class B : public A
{
  public:
    ~B() override {}                      // Compliant
    virtual void F() noexcept override    // Non-compliant - function is specified
                                          // with virtual and override
```

```
35      {
36      }
37      void K() noexcept override
38          final  // Non-compliant - function is specified with override and final
39      {
40      }
41      virtual void M() noexcept  // Compliant - violates A10-3-2
42      {
43      }
44      void Z() noexcept override  // Compliant
45      {
46      }
47      void J() noexcept  // Non-compliant - virtual function but not marked as
48                         // overrider
49      {
50      }
51      A& operator+=(A const& rhs) noexcept override  // Compliant - to override
52                                                     // the operator correctly,
53                                                     // its signature needs to be
54                                                     // the same as in the base
55                                                     // class
56      {
57          // ...
58          return *this;
59      }
60  };
```

See also

- C++ Core Guidelines [11]: C.128: Virtual functions should specify exactly one of virtual, override, or final.

Rule A10-3-2 (required, implementation, automated)
Each overriding virtual function shall be declared with the override or final specifier.

Rationale

Explicit use of the override or final specifier enables the compiler to catch mismatch of types and names between base and derived classes virtual functions.

Note that this rule applies to virtual destructor overriders, too.

Also, note that this rule applies to a pure virtual function which overrides another pure virtual function.

Example

```
1  // $Id: A10-3-2.cpp 289436 2017-10-04 10:45:23Z michal.szczepankiewicz $
2  class A
3  {
```

```
4     public:
5       virtual ~A() {}
6       virtual void F() noexcept = 0;
7       virtual void G() noexcept {}
8       virtual void Z() noexcept {}
9       virtual A& operator+=(A const& oth) = 0;
10  };
11  class B : public A
12  {
13    public:
14      ~B() override {}   // Compliant
15      void F() noexcept  // Non-compliant
16      {
17      }
18      virtual void G() noexcept  // Non-compliant
19      {
20      }
21      void Z() noexcept override  // Compliant
22      {
23      }
24      B& operator+=(A const& oth) override  // Compliant
25      {
26          return *this;
27      }
28  };
29  class C : public A
30  {
31    public:
32      ~C() {}                     // Non-compliant
33      void F() noexcept override  // Compliant
34      {
35      }
36      void G() noexcept override  // Compliant
37      {
38      }
39      void Z() noexcept override  // Compliant
40      {
41      }
42      C& operator+=(A const& oth)  // Non-compliant
43      {
44          return *this;
45      }
46  };
```

See also

- HIC++ v4.0 [9]: 10.2.1 Use the override special identifier when overriding a virtual function
- C++ Core Guidelines [11]: C.128: Virtual functions should specify exactly one of virtual, override, or final.

Rule A10-3-3 (required, implementation, automated)
Virtual functions shall not be introduced in a final class.

Rationale

Declaring a class as final explicitly specifies that the class cannot be inherited. Declaring a virtual function inside a class specifies that the function can be overridden in the inherited class, which is inconsistent.

Example

```
// $Id: A10-3-3.cpp 289436 2017-10-04 10:45:23Z michal.szczepankiewicz $
class A
{
  public:
    virtual ~A() = default;
    virtual void F() noexcept = 0;
    virtual void G() noexcept {}
};
class B final : public A
{
  public:
    void F() noexcept final  // Compliant
    {
    }
    void G() noexcept override  // Non-compliant
    {
    }
    virtual void H() noexcept = 0;  // Non-compliant
    virtual void Z() noexcept       // Non-compliant
    {
    }
};
```

See also

- HIC++ v4.0 [9]: 9.1.5 Do not introduce virtual functions in a final class.

Rule A10-3-5 (required, implementation, automated)
A user-defined assignment operator shall not be virtual.

Rationale

If an overloaded operator is declared virtual in a base class A, then in its subclasses B and C identical arguments list needs to be provided for the overriders. This allows to call an assignment operator of class B that takes an argument of type C which may lead to undefined behavior.

Note that this rule applies to all assignment operators, as well to copy and move assignment operators.

Example

```
// $Id: A10-3-4.cpp 289436 2017-10-04 10:45:23Z michal.szczepankiewicz $
class A
{
  public:
    virtual A& operator=(A const& oth) = 0;    // Non-compliant
    virtual A& operator+=(A const& rhs) = 0;   // Non-compliant
};
class B : public A
{
  public:
    B& operator=(A const& oth) override  // It needs to take an argument of type
                                         // A& in order to override
    {
        return *this;
    }
    B& operator+=(A const& oth) override  // It needs to take an argument of
                                          // type A& in order to override
    {
        return *this;
    }
    B& operator-=(B const& oth)  // Compliant
    {
        return *this;
    }
};
class C : public A
{
  public:
    C& operator=(A const& oth) override  // It needs to take an argument of type
                                         // A& in order to override
    {
        return *this;
    }
    C& operator+=(A const& oth) override  // It needs to take an argument of
                                          // type A& in order to override
    {
        return *this;
    }
    C& operator-=(C const& oth)  // Compliant
    {
        return *this;
    }
};
// class D : public A
//{
// public:
//    D& operator=(D const& oth) override  // Compile time error - this method
//    does not override because of different
//    signature
//    {
```

```
//         return *this;
//     }
//     D& operator+=(D const& oth) override  // Compile time error - this method
//     does not override because of different
//     signature
//     {
//         return *this;
//     }
//};
void Fn() noexcept
{
    B b;
    C c;
    b = c;   // Calls B::operator= and accepts an argument of type C
    b += c;  // Calls B::operator+= and accepts an argument of type C
    c = b;   // Calls C::operator= and accepts an argument of type B
    c += b;  // Calls C::operator+= and accepts an argument of type B
    // b -= c;  // Compilation error, because of types mismatch. Expected
    // behavior
    // c -= b;  // Compilation error, because of types mismatch. Expected
    // behavior

    B b2;
    C c2;
    b -= b2;
    c -= c2;
}
```

See also

- none

Rule M10-3-3 (required, implementation, automated)
A virtual function shall only be overridden by a pure virtual function if it is itself declared as pure virtual.

See MISRA C++ 2008 [7]

See: A10-3-2 for pure virtual function overriders declaration.

6.10.4 Abstract Classes

Rule A10-4-1 (advisory, design, non-automated)
Hierarchies should be based on interface classes.

Rationale

Software design that provides common and standardized interfaces without committing to a particular implementation:

- eliminates of potential redundancy.
- increases software reusability.
- hides implementation details.
- can be easily extended.
- facilitates different objects iteration.

Well-defined interfaces are less prone to require further reworking and maintenance.

See: `Interface-Class`

See also

- JSF December 2005 [8]: AV Rule 87: Hierarchies should be based on abstract classes.
- C++ Core Guidelines [11]: I.25: Prefer abstract classes as interfaces to class hierarchies.
- C++ Core Guidelines [11]: C.122: Use abstract classes as interfaces when complete separation of interface and implementation is needed.

6.11 Member access control

6.11.0 General

Rule M11-0-1 (required, implementation, automated)
Member data in non-POD class types shall be private.

See MISRA C++ 2008 [7]

See: `POD-type`, `Standard-Layout-Class`, `Trivially-Copyable`

Rule A11-0-1 (advisory, implementation, automated)
A non-POD type should be defined as class.

Rationale

Types that are not POD types are supposed to be defined as class objects, as a class specifier forces the type to provide private access control for all its members by default. This is consistent with developer expectations, because it is expected that a class has its invariant, interface and could provide custom-defined constructors.

Example

```
// $Id: A11-0-1.cpp 289436 2017-10-04 10:45:23Z michal.szczepankiewicz $
#include <cstdint>
#include <limits>
class A  // Compliant - A provides user-defined constructors, invariant and
         // interface
{
    std::int32_t x;  // Data member is private by default

  public:
    static constexpr std::int32_t maxValue =
        std::numeric_limits<std::int32_t>::max();
    A() : x(maxValue) {}
    explicit A(std::int32_t number) : x(number) {}
    A(A const&) = default;
    A(A&&) = default;
    A& operator=(A const&) = default;
    A& operator=(A&&) = default;

    std::int32_t GetX() const noexcept { return x; }
    void SetX(std::int32_t number) noexcept { x = number; }
};
struct B  // Non-compliant - non-POD type defined as struct
{
  public:
    static constexpr std::int32_t maxValue =
        std::numeric_limits<std::int32_t>::max();
    B() : x(maxValue) {}
    explicit B(std::int32_t number) : x(number) {}
    B(B const&) = default;
    B(B&&) = default;
    B& operator=(B const&) = default;
    B& operator=(B&&) = default;

    std::int32_t GetX() const noexcept { return x; }
    void SetX(std::int32_t number) noexcept { x = number; }

  private:
    std::int32_t x;  // Need to provide private access specifier for x member
};
struct C  // Compliant - POD type defined as struct
{
    std::int32_t x;
    std::int32_t y;
};
class D  // Compliant - POD type defined as class, but not compliant with
         // M11-0-1
{
  public:
    std::int32_t x;
    std::int32_t y;
};
```

See also

- C++ Core Guidelines [11]: C.2: Use class if the class has an invariant; use struct if the data members can vary independently.
- stackoverflow.com [17]: When should you use a class vs a struct in C++?

Rule A11-0-2 (required, implementation, automated)
A type defined as struct shall: (1) provide only public data members, (2) not provide any special member functions or methods, (3) not be a base of another struct or class, (4) not inherit from another struct or class.

Rationale

This is consistent with developer expectations that a class provides its invariant, interface and encapsulation guarantee, while a struct is only an aggregate without any class-like features.

An example of a struct type is POD type.

See: POD-type.

Example

```
// $Id: A11-0-2.cpp 289436 2017-10-04 10:45:23Z michal.szczepankiewicz $
#include <cstdint>
struct A  // Compliant
{
    std::int32_t x;
    double y;
};
struct B  // Compliant
{
    std::uint8_t x;
    A a;
};
struct C  // Compliant
{
    float x = 0.0f;
    std::int32_t y = 0;
    std::uint8_t z = 0U;
};
struct D  // Non-compliant
{
  public:
    std::int32_t x;

  protected:
    std::int32_t y;

  private:
```

```
28      std::int32_t z;
29  };
30  struct E  // Non-compliant
31  {
32    public:
33      std::int32_t x;
34      void Fn() noexcept {}
35
36    private:
37      void F1() noexcept(false) {}
38  };
39  struct F : public D  // Non-compliant
40  {
41  };
```

See also

- stackoverflow.com [17]: When should you use a class vs a struct in C++?

6.11.3 Friends

Rule A11-3-1 (required, implementation, automated)
Friend declarations shall not be used.

Rationale

Friend declarations reduce encapsulation and result in code that is more difficult to maintain.

Exception

It is allowed to declare comparison operators as friend functions, see A13-5-5.

Example

```
1   // $Id: A11-3-1.cpp 325916 2018-07-13 12:26:22Z christof.meerwald $
2   class A
3   {
4     public:
5       A& operator+=(A const& oth);
6       friend A const operator+(A const& lhs, A const& rhs);  // Non-compliant
7   };
8   class B
9   {
10    public:
11      B& operator+=(B const& oth);
12      friend bool operator ==(B const& lhs, B const& rhs)  // Compliant by
        exception
13      {
```

```
14          // Implementation
15      }
16  };
17
18  B const operator+(B const& lhs, B const& rhs)  // Compliant
19  {
20      // Implementation
21  }
```

See also

- JSF December 2005 [8]: AV Rule 70 A class will have friends only when a function or object requires access to the private elements of the class, but is unable to be a member of the class for logical or efficiency reasons.
- HIC++ v4.0 [9]: 11.2.1 Do not use friend declarations.

6.12 Special member functions

6.12.0 General

Rule A12-0-1 (required, implementation, automated)
If a class declares a copy or move operation, or a destructor, either via "=default", "=delete", or via a user-provided declaration, then all others of these five special member functions shall be declared as well.

Rationale

The semantics of five of the special member functions,

- the copy constructor,
- the move constructor,
- the copy assignment operator,
- the move assignment operator,
- and the destructor,

are closely related to each other. If, for example, there is need to provide a non-default destructor to release a resource, special handling usually also needs to be added in the copy and move operations to properly handle this resource.

Language rules exist to generate the special member functions under certain conditions. For historical reasons, these language rules are not entirely consistent. For example, the presence of a destructor does not prevent the compiler from generating copy operations. However, it prevents the move operations from being generated. Thus

it is required, in order to maintain consistency and document the programmer's intent, that either none or all of the five functions are declared.

This rule is also known as "the rule of zero", or "the rule of five" respectively. It is highly recommended to design classes in a way that the rule of zero can be followed.

Note that the default constructor (which is also a special member function) is not part of this rule. The presence of any user-declared constructor inhibits the generation of the default constructor. Therefore, if a user-declared constructor is present, it may be necessary (depending on requirements) to also declare the default constructor. However, the presence of a user-declared default constructor does not inhibit the generation of the other five special member functions. This rule therefore allows to follow the rule of zero when the class only has a user-declared default constructor (and possibly one or more constructors which are not special member functions).

Example

```
1  // $Id: A12-0-1.cpp 309769 2018-03-01 17:40:29Z jan.babst $
2  #include <string>
3
4  namespace v1
5  {
6  // Class is copyable and moveable via the compiler generated funtions.
7  // Compliant - rule of zero.
8  class A
9  {
10   private:
11     // Member data ...
12 };
13 }  // namespace v1
14
15 namespace v2
16 {
17 // New requirement: Destructor needs to be added.
18 // Now the class is no longer moveable, but still copyable. The program
19 // still compiles, but may perform worse.
20 // Non-compliant - Unclear if this was the developers intent.
21 class A
22 {
23   public:
24     ~A()
25     {
26         // ...
27     }
28
29   private:
30     // Member data ...
31 };
32 }  // namespace v2
33
34 namespace v3
35 {
```

```
// Move operations are brought back by defaulting them.
// Copy operations are defaulted since they are no longer generated
// (complies to A12-0-1 but will also be a compiler error if they are needed).
// Default constructor is defaulted since it is no longer generated
// (not required by A12-0-1 but will be a compiler error if it is needed).
// Compliant - rule of five. Programmer's intent is clear, class behaves the
// same as v1::A.
class A
{
  public:
    A() = default;
    A(A const&) = default;
    A(A&&) = default;
    ~A()
    {
        // ...
    }
    A& operator=(A const&) = default;
    A& operator=(A&&) = default;

  private:
    // Member data ...
};
}  // namespace v3

// A class with regular (value) semantics.
// Compliant - rule of zero.
class Simple
{
  public:
    // User defined constructor, also acts as default constructor.
    explicit Simple(double d = 0.0, std::string s = "Hello")
            : d_(d), s_(std::move(s))
    {
    }

    // Compiler generated copy c'tor, move c'tor, d'tor, copy assignment, move
    // assignment.

  private:
    double d_;
    std::string s_;
};

// A base class.
// Compliant - rule of five.
class Base
{
  public:
    Base(Base const&) = delete;             // see also A12-8-6
    Base(Base&&) = delete;                  // see also A12-8-6
```

```
87      virtual ~Base() = default;              // see also A12-4-1
88      Base& operator=(Base const&) = delete;  // see also A12-8-6
89      Base& operator=(Base&&) = delete;       // see also A12-8-6
90
91      // Declarations of pure virtual functions ...
92
93    protected:
94      Base() = default;  // in order to allow construction of derived objects
95  };
96
97  // A move-only class.
98  // Compliant - rule of five.
99  class MoveOnly
100 {
101   public:
102     MoveOnly();
103     MoveOnly(MoveOnly const&) = delete;
104     MoveOnly(MoveOnly&&) noexcept;
105     ~MoveOnly();
106     MoveOnly& operator=(MoveOnly const&) = delete;
107     MoveOnly& operator=(MoveOnly&&) noexcept;
108
109   private:
110     // ...
111 };
```

See also

- C++ Core Guidelines [11]: C.21: If you define or =delete any default operation, define or =delete them all.
- C++ Core Guidelines [11]: C.81: Use =delete when you want to disable default behavior (without wanting an alternative).

Rule A12-0-2 (required, implementation, partially automated)
Bitwise operations and operations that assume data representation in memory shall not be performed on objects.

Rationale

Object representations may consist of more than only the declared fields (unless the objects are standard-layout or trivially copyable). Performing bitwise operations on objects may access bits that are not part of the value representation, which may lead to undefined behavior. Operations on objects (e.g. initialization, copying, comparing, setting, accessing) shall be done by dedicated constructors, overloaded operators, accessors or mutators.

Example

```
1  // $Id: A12-0-2.cpp 305629 2018-01-29 13:29:25Z piotr.serwa $
```

```
//
#include <cstdint>
#include <cstring>

class A
{
public:
    A() = default;
    A(uint8_t c, uint32_t i, int8_t d, int32_t h) : c(c), i(i), d(d), h(h) {}

    bool operator==(const A& rhs) const noexcept
    {
        return c==rhs.c && i==rhs.i && d==rhs.d && h==rhs.h;
    }

private:
    uint8_t c;
    uint32_t i;
    int8_t d;
    int32_t h;
};


int main(void)
{
    A noninit;

    //setting field c
    std::memset(&noninit, 3, 1); //non-compliant
    //setting field i
    std::memset(((uint8_t*)&noninit)+sizeof(uint8_t)+3, 5, 1);  //non-compliant

    A init(3, 5, 7, 9); //compliant

    if (noninit == init) //compliant
    {

    }

    if (0 == std::memcmp(&noninit, &init, sizeof(init)))
    { //non-compliant, includes padding bytes

    }

    return 0;
}
```

See also

- JSF December 2005 [8]: AV Rule 156: All the members of a structure (or class) shall be named and shall only be accessed via their names.

- JSF December 2005 [8]: AV Rule 210: Algorithms shall not make assumptions concerning how data is represented in memory (e.g. big endian vs. little endian, base class subobject ordering in derived classes, nonstatic data member ordering across access specifiers, etc.)
- JSF December 2005 [8]: AV Rule 210.1: Algorithms shall not make assumptions concerning the order of allocation of nonstatic data members separated by an access specifier.
- JSF December 2005 [8]: AV Rule 211: Algorithms shall not assume that shorts, ints, longs, floats, doubles or long doubles begin at particular addresses.
- SEI CERT C++ Coding Standard [10]: EXP42-C: Do not compare padding data
- SEI CERT C++ Coding Standard [10]: EXP62-C: Do not access the bits of an object representation that are not part of the object's value representation
- SEI CERT C++ Coding Standard [10]: OOP57-CPP: Prefer special member functions and overloaded operators to C Standard Library functions

6.12.1 Constructors

Rule A12-1-1 (required, implementation, automated)
Constructors shall explicitly initialize all virtual base classes, all direct non-virtual base classes and all non-static data members.

Rationale

A constructor of a class is supposed to completely initialize its object. Explicit initialization of all virtual base classes, direct non-virtual base classes and non-static data members reduces the risk of an invalid state after successful construction.

Example

```
1  // $Id: A12-1-1.cpp 271696 2017-03-23 09:23:09Z piotr.tanski $
2  #include <cstdint>
3  class Base
4  {
5      // Implementation
6  };
7  class VirtualBase
8  {
9  };
10 class A : public virtual VirtualBase, public Base
11 {
12   public:
13     A() : VirtualBase{}, Base{}, i{0}, j{0}  // Compliant
14     {
15     }
```

```
16      A(A const& oth)
17              : Base{}, j{0}  // Non-compliant - VirtualBase base class and member
18                              // i not initialized
19      {
20      }
21
22    private:
23      std::int32_t i;
24      std::int32_t j;
25      static std::int32_t k;
26  };
27  std::int32_t A::k{0};
```

See also

- MISRA C++ 2008 [7]: Rule 12-1-2 All constructors of a class should explicitly call a constructor for all of its immediate base classes and all virtual base classes.
- HIC++ v4.0 [9]:12.4.2 Ensure that a constructor initializes explicitly all base classes and non-static data members.
- JSF December 2005 [8]: AV Rule 71: Calls to an externally visible operation of an object, other than its constructors, shall not be allowed until the object has been fully initialized.

Rule M12-1-1 (required, implementation, automated)
An object's dynamic type shall not be used from the body of its constructor or destructor.

See MISRA C++ 2008 [7]

Note: This rule prohibits both direct and indirect usage of object's dynamic type from its constructor or destructor.

See also

- C++ Core Guidelines [11]: C.50: Use a factory function if you need "virtual behavior" during initialization.

Rule A12-1-2 (required, implementation, automated)
Both NSDMI and a non-static member initializer in a constructor shall not be used in the same type.

Rationale

Since 2011 C++ Language Standard it is allowed to initialize a non-static member along with the declaration of the member in the class body using NSDMI ("non-static data member initializer"). To avoid possible confusion which values are actually used,

if any member is initialized by NSDMI or with a constructor, then all others should be initialized the same way.

Exception

The move and copy constructors are exempt from this rule, because these constructors copy the existing values from other objects.

Example

```
// $Id: A12-1-2.cpp 271696 2017-03-23 09:23:09Z piotr.tanski $
#include <cstdint>
#include <utility>
class A
{
  public:
    A() : i1{0}, i2{0}  // Compliant - i1 and i2 are initialized by the
                        // constructor only. Not compliant with A12-1-3
    {
    }
    // Implementation

  private:
    std::int32_t i1;
    std::int32_t i2;
};
class B
{
  public:
    // Implementation

  private:
    std::int32_t i1{0};
    std::int32_t i2{
        0};  // Compliant - both i1 and i2 are initialized by NSDMI only
};
class C
{
  public:
    C() : i2{0}  // Non-compliant - i1 is initialized by NSDMI, i2 is in
                 // member in member initializer list
    {
    }
    C(C const& oth) : i1{oth.i1}, i2{oth.i2}  // Compliant by exception
    {
    }
    C(C&& oth)
            : i1{std::move(oth.i1)},
              i2{std::move(oth.i2)}  // Compliant by exception
    {
    }
    // Implementation
```

```
43
44    private:
45      std::int32_t i1{0};
46      std::int32_t i2;
47  };
```

See also

- HIC++ v4.0 [9]:12.4.3 Do not specify both an NSDMI and a member initializer in a constructor for the same non static member

Rule A12-1-3 (required, implementation, automated)
If all user-defined constructors of a class initialize data members with constant values that are the same across all constructors, then data members shall be initialized using NSDMI instead.

Rationale

Using NSDMI lets the compiler to generate the function that can be more efficient than a user-defined constructor that initializes data member variables with pre-defined constant values.

Example

```
1  // $Id: A12-1-3.cpp 291949 2017-10-19 21:26:22Z michal.szczepankiewicz $
2  #include <cstdint>
3  #include <string>
4  class A
5  {
6    public:
7      A() : x(0), y(0.0F), str()  // Non-compliant
8      {
9      }
10     // ...
11
12   private:
13     std::int32_t x;
14     float y;
15     std::string str;
16 };
17 class B
18 {
19   public:
20     // ...
21
22   private:
23     std::int32_t x = 0;     // Compliant
24     float y = 0.0F;         // Compliant
25     std::string str = "";   // Compliant
26 };
```

```
class C
{
  public:
    C() : C(0, 0.0F, decltype(str)())  // Compliant
    {
    }
    C(std::int32_t i, float f, std::string s) : x(i), y(f), str(s)  // Compliant
    {
    }
    // ...

  private:
    std::int32_t x =
        0;           // Non-compliant - there's a constructor that initializes C
                     // class with user input
    float y = 0.0F;  // Non-compliant - there's a constructor that initializes C
                     // class with user input
    std::string str = "";  // Non-compliant - there's a constructor that
                           // initializes C class with user input
};
```

See also

- C++ Core Guidelines [11]: C.45: Don't define a default constructor that only initializes data members; use in-class member initializers instead.

Rule A12-1-4 (required, implementation, automated)
All constructors that are callable with a single argument of fundamental type shall be declared explicit.

Rationale

The explicit keyword prevents the constructor from being used to implicitly convert a fundamental type to the class type.

See: Fundamental-Types.

Example

```
// $Id: A12-1-4.cpp 289436 2017-10-04 10:45:23Z michal.szczepankiewicz $
#include <cstdint>
class A
{
  public:
    explicit A(std::int32_t number) : x(number) {}  // Compliant
    A(A const&) = default;
    A(A&&) = default;
    A& operator=(A const&) = default;
    A& operator=(A&&) = default;

```

```
  private:
    std::int32_t x;
};
class B
{
  public:
    B(std::int32_t number) : x(number) {}  // Non-compliant
    B(B const&) = default;
    B(B&&) = default;
    B& operator=(B const&) = default;
    B& operator=(B&&) = default;

  private:
    std::int32_t x;
};
void F1(A a) noexcept
{
}
void F2(B b) noexcept
{
}
void F3() noexcept
{
    F1(A(10));
    // f1(10);  // Compilation error - because of explicit constructor it is not
    // possible to implicitly convert integer
    // to type of class A
    F2(B(20));
    F2(20);  // No compilation error - implicit conversion occurs
}
```

See also

- MISRA C++ 2008 [7]: Rule 12-1-3 (Required) All constructors that are callable with a single argument of fundamental type shall be declared explicit.

Rule A12-1-5 (required, implementation, partially automated)
Common class initialization for non-constant members shall be done by a delegating constructor.

Rationale

Common initialization of non-constant members in a delegating constructor prevents from code repetition, accidental differences and maintenance problems.

Example

```
// $Id: A12-1-5.cpp 289436 2017-10-04 10:45:23Z michal.szczepankiewicz $

#include <cstdint>
```

```
4
5  class A
6  {
7    public:
8      // Compliant
9      A(std::int32_t x, std::int32_t y) : x(x + 8), y(y) {}
10     explicit A(std::int32_t x) : A(x, 0) {}
11
12   private:
13     std::int32_t x;
14     std::int32_t y;
15 };
16
17 class B
18 {
19   public:
20     // Non-compliant
21     B(std::int32_t x, std::int32_t y) : x(x + 8), y(y) {}
22     explicit B(std::int32_t x) : x(x + 8), y(0) {}
23
24   private:
25     std::int32_t x;
26     std::int32_t y;
27 };
```

See also

- HIC++ v4.0 [9]: 12.4.5: Use delegating constructors to reduce code duplication.
- C++ Core Guidelines [11]: C.51: Use delegating constructors to represent common actions for all constructors of a class.

Rule A12-1-6 (required, implementation, automated)
Derived classes that do not need further explicit initialization and require all the constructors from the base class shall use inheriting constructors.

Rationale

Reimplementing constructors that do not need further initialization is error-prone and may lead to using wrong base class constructor accidentally.

Example

```
1  // $Id: A12-1-6.cpp 289436 2017-10-04 10:45:23Z michal.szczepankiewicz $
2
3  #include <cstdint>
4
5  class A
6  {
7    public:
8      A(std::int32_t x, std::int32_t y) : x(x + 8), y(y) {}
```

```
9       explicit A(std::int32_t x) : A(x, 0) {}
10
11    private:
12      std::int32_t x;
13      std::int32_t y;
14  };
15
16  // Non-compliant
17  class B : public A
18  {
19    public:
20      B(std::int32_t x, std::int32_t y) : A(x, y) {}
21      explicit B(std::int32_t x) : A(x) {}
22  };
23
24  // Compliant
25  class C : public A
26  {
27    public:
28      using A::A;
29  };
```

See also

- C++ Core Guidelines [11]: C.52: Use inheriting constructors to import constructors into a derived class that does not need further explicit initialization.

6.12.4 Destructors

Rule A12-4-1 (required, implementation, automated)
Destructor of a base class shall be public virtual, public override or protected non-virtual.

Rationale

If an object is supposed to be destroyed through a pointer or reference to its base class, the destructor in the base class needs to be virtual. Otherwise, destructors for derived types will not be invoked.

Note that if it is prohibited to destroy an object through a pointer or reference to its base class, the destructor in the base class is supposed to be protected.

Example

```
1  // $Id: A12-4-1.cpp 289436 2017-10-04 10:45:23Z michal.szczepankiewicz $
2  class A
3  {
4    public:
5      ~A()  // Non-compliant
```

```
6       {
7       }
8   };
9   class B : public A
10  {
11  };
12  class C
13  {
14    public:
15      virtual ~C()  // Compliant
16      {
17      }
18  };
19  class D : public C
20  {
21  };
22  class E
23  {
24    protected:
25      ~E();  // Compliant
26  };
27  class F : public E
28  {
29  };
30  void F1(A* obj1, C* obj2)
31  {
32      // ...
33      delete obj1;  // Only destructor of class A will be invoked
34      delete obj2;  // Both destructors of D and C will be invoked
35  }
36  void F2()
37  {
38      A* a = new B;
39      C* c = new D;
40      F1(a, c);
41  }
```

See also

- JSF December 2005 [8]: AV Rule 78 All base classes with a virtual function shall define a virtual destructor.
- HIC++ v4.0 [9]: 12.2.1 Declare virtual, private or protected the destructor of a type used as a base class.
- SEI CERT C++ Coding Standard [10]: OOP52-CPP: Do not delete a polymorphic object without a virtual destructor.
- C++ Core Guidelines [11]: C.35: A base class destructor should be either public and virtual, or protected and nonvirtual.

- C++ Core Guidelines [11]: Discussion: Make base class destructors public and virtual, or protected and nonvirtual.

Rule A12-4-2 (advisory, implementation, automated)
If a public destructor of a class is non-virtual, then the class should be declared final.

Rationale

If a public destructor of a class is non-virtual (i.e. no virtual, override or final keyword), then the class is not supposed to be used as a base class in inheritance hierarchy.

Note that a destructor needs to be virtual in a base class in order to correctly destroy an instance of a derived class through a pointer to the base class.

Example

```
// $Id: A12-4-2.cpp 289436 2017-10-04 10:45:23Z michal.szczepankiewicz $
class A  // Non-compliant - class A should not be used as a base class because
         // its destructor is not virtual, but it is
         // not declared final
{
  public:
    A() = default;
    A(A const&) = default;
    A(A&&) = default;
    A& operator=(A const&) = default;
    A& operator=(A&&) = default;
    ~A() = default;  // Public non-virtual destructor
};
class B final  // Compliant - class B can not be used as a base class, because
               // it is declared final, and it should not be derived
               // because its destructor is not virtual
{
  public:
    B() = default;
    B(B const&) = default;
    B(B&&) = default;
    B& operator=(B const&) = default;
    B& operator=(B&&) = default;
    ~B() = default;  // Public non-virtual destructor
};
class C  // Compliant - class C is not final, and its destructor is virtual. It
         // can be used as a base class
{
  public:
    C() = default;
    C(C const&) = default;
    C(C&&) = default;
    C& operator=(C const&) = default;
    C& operator=(C&&) = default;
```

```
35      virtual ~C() = default;  // Public virtual destructor
36  };
37  class AA : public A
38  {
39  };
40  // class BA : public B  // Compilation error - can not derive from final base
41  // class B
42  //{
43  //};
44  class CA : public C
45  {
46  };
47  void Fn() noexcept
48  {
49      AA obj1;
50      CA obj2;
51      A& ref1 = obj1;
52      C& ref2 = obj2;
53
54      ref1.~A();  // Calls A::~A() only
55      ref2.~C();  // Calls both CA::~CA() and C::~C()
56  }
```

See also

- SEI CERT C++ Coding Standard [10]: OOP52-CPP: Do not delete a polymorphic object without a virtual destructor.

6.12.6 Initialization

Rule A12-6-1 (required, implementation, automated)
All class data members that are initialized by the constructor shall be initialized using member initializers.

Rationale

Using the constructor's member initializers is more efficient than assigning a copy of passed values to data members in the constructor's body. Also, it supports the programmer to prevent "data usage before initialization" errors.

Note that if a data member is already initialized using member initializer, then changing its value in the constructor's body does not violate this rule.

Example

```
1  // $Id: A12-6-1.cpp 271696 2017-03-23 09:23:09Z piotr.tanski $
2  #include <cstdint>
3  #include <string>
4  class A
```

```
{
  public:
    A(std::int32_t n, std::string s) : number{n}, str{s}  // Compliant
    {
    }
    // Implementation

  private:
    std::int32_t number;
    std::string str;
};
class B
{
  public:
    B(std::int32_t n, std::string s)  // Non-compliant - no member initializers
    {
        number = n;
        str = s;
    }
    // Implementation

  private:
    std::int32_t number;
    std::string str;
};
class C
{
  public:
    C(std::int32_t n, std::string s) : number{n}, str{s}  // Compliant
    {
        n += 1;  // This does not violate the rule
        str.erase(str.begin(),
                  str.begin() + 1);  // This does not violate the rule
    }
    // Implementation

  private:
    std::int32_t number;
    std::string str;
};
```

See also

- C++ Core Guidelines [11]: C.49: Prefer initialization to assignment in constructors.

6.12.7 Construction and destructions

Rule A12-7-1 (required, implementation, automated)
If the behavior of a user-defined special member function is identical to implicitly defined special member function, then it shall be defined "=default" or be left undefined.

Rationale

If a user-defined version of a special member function is the same as would be provided by the compiler, it will be less error prone and more maintainable to replace it with "=default" definition or leave it undefined to let the compiler define it implicitly.

Note that this rule applies to all special member functions of a class.

See: Implicitly-Defined-Default-Constructor, Implicitly-Defined-Copy-Constructor, Implicitly-Defined-Move-Constructor, Implicitly-Defined-Copy-Assignment-Operator, Implicitly-Defined-Move-Assignment-Operator, Implicitly-Defined-Destructor

Example

```
1  // $Id: A12-7-1.cpp 271715 2017-03-23 10:13:51Z piotr.tanski $
2  #include <cstdint>
3  #include <utility>
4  class A
5  {
6    public:
7      A() : x(0), y(0) {}  // Compliant
8      A(std::int32_t first, std::int32_t second) : x(first), y(second) {}  //
       Compliant
9                                                                          // -
10     // anyway, such
11     // a constructor
12     // cannot be
13     // defaulted.
14     A(const A& oth)
15             : x(oth.x),
16               y(oth.y)  // Non-compliant - equivalent to the implicitly
17                         // defined copy constructor
18     {
19     }
20     A(A&& oth)
21             : x(std::move(oth.x)),
22               y(std::move(
23                   oth.y))  // Non-compliant - equivalent to the implicitly
24                            // defined move constructor
25     {
26     }
27     ~A()  // Non-compliant - equivalent to the implicitly defined destructor
28     {
29     }
30
```

```
  private:
    std::int32_t x;
    std::int32_t y;
};
class B
{
  public:
    B() {}  // Non-compliant - x and y are not initialized
            // should be replaced with: B() : x{0}, y{0} {}
    B(std::int32_t first, std::int32_t second) : x(first), y(second) {}  // Compliant
    B(const B&) =
        default;  // Compliant - equivalent to the copy constructor of class A
    B(B&&) =
        default;  // Compliant - equivalent to the move constructor of class A
    ~B() = default;  // Compliant - equivalent to the destructor of class A

  private:
    std::int32_t x;
    std::int32_t y;
};
class C
{
  public:
    C() = default;           // Compliant
    C(const C&) = default;   // Compliant
    C(C&&) = default;        // Compliant
};
class D
{
  public:
    D() : ptr(nullptr) {}    // Compliant - this is not equivalent to what the
                             // implicitly defined default constructor would do
    D(C* p) : ptr(p) {}      // Compliant
    D(const D&) = default;   // Shallow copy will be performed, user-defined copy
    // constructor is needed to perform deep copy on ptr variable
    D(D&&) = default;  // ptr variable will be moved, so ptr will still point to
                       // the same object
    ~D() = default;  // ptr will not be deleted, the user-defined destructor is
                     // needed to delete allocated memory

  private:
    C* ptr;
};
class E  // Compliant - special member functions definitions are not needed as
         // class E uses only implicit definitions
{
};
```

See also

- HIC++ v4.0 [9]: 12.5.2 Define special members =default if the behavior is equivalent.
- C++ Core Guidelines [11]: C.80: Use =default if you have to be explicit about using the default semantics.

6.12.8 Copying and moving class objects

Rule A12-8-1 (required, implementation, automated)
Move and copy constructors shall move and respectively copy base classes and data members of a class, without any side effects.

Rationale

It is expected behavior that the move/copy constructors are only used to move/copy the object of the class type and possibly set moved-from object to a valid state.

Move and copy constructors of an object are frequently called by STL algorithms and containers, so they are not supposed to provide any performance overhead or side effects that could affect moving or copying the object.

Note: Class members that are not essential for a class invariant may not need to be copied (e.g. caches, debug information).

Example

```
1  // $Id: A12-8-1.cpp 303582 2018-01-11 13:42:56Z michal.szczepankiewicz $
2  #include <cstdint>
3  #include <utility>
4  class A
5  {
6    public:
7      // Implementation
8      A(A const& oth) : x(oth.x)  // Compliant
9      {
10     }
11
12   private:
13     std::int32_t x;
14 };
15 class B
16 {
17   public:
18     // Implementation
19     B(B&& oth) : ptr(std::move(oth.ptr))  // Compliant
20     {
21         oth.ptr = nullptr;  // Compliant - this is not a side-effect, in this
22                             // case it is essential to leave moved-from object
23                             // in a valid state, otherwise double deletion will
```

```
24                                  // occur.
25      }
26      ~B() { delete ptr; }
27
28    private:
29      std::int32_t* ptr;
30  };
31  class C
32  {
33    public:
34      // Implementation
35      C(C const& oth) : x(oth.x)
36      {
37          // ...
38          x = x % 2;  // Non-compliant - unrelated side-effect
39      }
40
41    private:
42      std::int32_t x;
43  };
44
45  class D
46  {
47  public:
48      explicit D(std::uint32_t a) : a(a), noOfModifications(0) {}
49      D(const D& d) : D(d.a) {} //compliant, not copying the debug information
        about number of modifications
50      void SetA(std::uint32_t aa)
51      {
52          ++noOfModifications;
53          a = aa;
54      }
55      std::uint32_t GetA() const noexcept
56      {
57          return a;
58      }
59
60  private:
61      std::uint32_t a;
62      std::uint64_t noOfModifications;
63  };
```

See also

- MISRA C++ 2008 [7]: Rule 12-8-1 A copy constructor shall only initialize its base classes and the nonstatic members of the class of which it is a member.
- HIC++ v4.0 [9]: 12.5.3 Ensure that a user defined move/copy constructor only moves/copies base and member objects.

- JSF December 2005 [8]: AV Rule 77: A copy constructor shall copy all data members and bases that affect the class invariant (a data element representing a cache, for example, would not need to be copied).

Rule A12-8-2 (advisory, implementation, automated)
User-defined copy and move assignment operators should use user-defined no-throw swap function.

Rationale

Using a non-throwing swap operation in the copy and move assignment operators helps to achieve Strong Exception Safety. Each assignment operator is also simplified because it does not require check for assignment to itself.

Example

```
// $Id: A12-8-2.cpp 289436 2017-10-04 10:45:23Z michal.szczepankiewicz $
#include <cstdint>
#include <utility>
class A
{
  public:
    A(const A& oth)
    {
        // ...
    }
    A(A&& oth) noexcept
    {
        // ...
    }
    A& operator=(const A& oth) &  // Compliant
    {
        A tmp(oth);
        Swap(*this, tmp);
        return *this;
    }
    A& operator=(A&& oth) & noexcept  // Compliant
    {
        A tmp(std::move(oth));
        Swap(*this, tmp);
        return *this;
    }
    static void Swap(A& lhs, A& rhs) noexcept
    {
        std::swap(lhs.ptr1, rhs.ptr1);
        std::swap(lhs.ptr2, rhs.ptr2);
    }

  private:
    std::int32_t* ptr1;
```

```
35      std::int32_t* ptr2;
36  };
37  class B
38  {
39    public:
40      B& operator=(const B& oth) &  // Non-compliant
41      {
42          if (this != &oth)
43          {
44              ptr1 = new std::int32_t(*oth.ptr1);
45              ptr2 = new std::int32_t(
46                  *oth.ptr2);  // Exception thrown here results in
47                               // a memory leak of ptr1
48          }
49
50          return *this;
51      }
52      B& operator=(B&& oth) & noexcept  // Non-compliant
53      {
54          if (this != &oth)
55          {
56              ptr1 = std::move(oth.ptr1);
57              ptr2 = std::move(oth.ptr2);
58              oth.ptr1 = nullptr;
59              oth.ptr2 = nullptr;
60          }
61
62          return *this;
63      }
64
65    private:
66      std::int32_t* ptr1;
67      std::int32_t* ptr2;
68  };
```

See also

- HIC++ v4.0 [9]: 12.5.6 Use an atomic, non-throwing swap operation to implement the copy and move assignment operators

Rule A12-8-3 (required, implementation, partially automated)
Moved-from object shall not be read-accessed.

Rationale

Except in rare circumstances, an object will be left in an unspecified state after its values has been moved into another object. Accessing data members of such object may result in abnormal behavior and portability concerns.

Exception

It is permitted to access internals of a moved-from object if it is guaranteed to be left in a well-specified state.

The following Standard Template Library functions are guaranteed to leave the moved-from object in a well-specified state:

- move construction, move assignment, “converting” move construction and “converting” move assignment of std::unique_ptr type
- move construction, move assignment, “converting” move construction, “converting” move assignment of std::shared_ptr type
- move construction and move assignment from a std::unique_ptr of std::shared_ptr type
- move construction, move assignment, “converting” move construction and “converting” move assignment of std::weak_ptr type
- std::move() of std::basic_ios type
- move constructor and move assignment of std::basic_filebuf type
- move constructor and move assignment of std::thread type
- move constructor and move assignment of std::unique_lock type
- move constructor and move assignment of std::shared_lock type
- move constructor and move assignment of std::promise type
- move constructor and move assignment of std::future type
- move construction, move assignment, “converting” move construction and “converting” move assignment of std::shared_future type
- move constructor and move assignment of std::packaged_task type

Example

```
1  // $Id: A12-8-3.cpp 289436 2017-10-04 10:45:23Z michal.szczepankiewicz $
2  #include <cstdint>
3  #include <iostream>
4  #include <memory>
5  #include <string>
6  void F1()
7  {
8      std::string s1{"string"};
9      std::string s2{std::move(s1)};
10     // ...
11     std::cout << s1 << "\n";  // Non-compliant - s1 does not contain "string"
12                               // value after move operation
13 }
14 void F2()
```

```
{
    std::unique_ptr<std::int32_t> ptr1 = std::make_unique<std::int32_t>(0);
    std::unique_ptr<std::int32_t> ptr2{std::move(ptr1)};
    std::cout << ptr1.get() << std::endl;  // Compliant by exception - move
                                           // construction of std::unique_ptr
                                           // leaves moved-from object in a
                                           // well-specified state
}
```

See also

- SEI CERT C++ [10]: EXP63-CPP Do not rely on the value of a moved-from object.

Rule A12-8-4 (required, implementation, automated)
Move constructor shall not initialize its class members and base classes using copy semantics.

Rationale

Data members or base classes initialization in move constructor needs to be done with move semantics. Move construction is an optimization strategy and the copy-initialization for data members and base classes will have negative impact on the program's performance, as well as it does not meet developer expectations.

Exception

In move constructor, copy initialization for data members of scalar types does not violate this rule.

See: Scalar-Types.

Example

```
// $Id: A12-8-4.cpp 271696 2017-03-23 09:23:09Z piotr.tanski $
#include <cstdint>
#include <string>
class A
{
  public:
    // ...
    A(A&& oth)
            : x(std::move(oth.x)),  // Compliant
              s(std::move(oth.s))   // Compliant
    {
    }

  private:
    std::int32_t x;
    std::string s;
};
```

```
18 class B
19 {
20   public:
21     // ...
22     B(B&& oth)
23             : x(oth.x),  // Compliant by exception, std::int32_t is of scalar
24                          // type
25               s(oth.s)   // Non-compliant
26     {
27     }
28
29   private:
30     std::int32_t x;
31     std::string s;
32 };
33 class C
34 {
35   public:
36     // ...
37     C(C&& oth)
38             : x(oth.x),            // Compliant by exception
39               s(std::move(oth.s))  // Compliant
40     {
41     }
42
43   private:
44     std::int32_t x = 0;
45     std::string s = "Default string";
46 };
```

See also

- SEI CERT C++ [10]: OOP11-CPP Do not copy-initialize members or base classes from a move constructor.

Rule A12-8-5 (required, implementation, automated)
A copy assignment and a move assignment operators shall handle self-assignment.

Rationale

User-defined copy assignment operator and move assignment operator need to prevent self-assignment, so the operation will not leave the object in an indeterminate state. If the given parameter is the same object as the local object, destroying object-local resources will invalidate them. It violates the copy/move assignment postconditions.

Note that STL containers assume that self-assignment of an object is correctly handled. Otherwise it may lead to unexpected behavior of an STL container.

Self-assignment problem can also be solved using swap operators. See rule: A12-8-2.

Example

```
// $Id: A12-8-5.cpp 271773 2017-03-23 13:16:53Z piotr.tanski $
#include <cstdint>
#include <stdexcept>
struct A
{
    std::int32_t number;
    std::int32_t* ptr;
    // Implementation
};
class B
{
  public:
    // ...
    B& operator=(B const& oth)  // Non-compliant
    {
        i = oth.i;
        delete aPtr;

        try
        {
            aPtr = new A(*oth.aPtr);  // If this is the self-copy case, then
                                      // the oth.a_ptr is already deleted
        }
        catch (std::bad_alloc&)
        {
            aPtr = nullptr;
        }

        return *this;
    }

  private:
    std::int16_t i = 0;
    A* aPtr = nullptr;
};
class C
{
  public:
    C& operator=(C const& oth)  // Compliant
    {
        if (this != &oth)
        {
            A* tmpPtr = new A(*oth.aPtr);

            i = oth.i;
            delete aPtr;
            aPtr = tmpPtr;
        }
```

```
49          return *this;
50      }
51      C& operator=(C&& oth)  // Compliant
52      {
53          if (this != &oth)
54          {
55              A* tmpPtr = new A{std::move(*oth.aPtr)};
56
57              i = oth.i;
58              delete aPtr;
59              aPtr = tmpPtr;
60          }
61          return *this;
62      }
63
64    private:
65      std::int16_t i = 0;
66      A* aPtr = nullptr;
67  };
```

See also

- SEI CERT C++ [10]: OOP54-CPP Gracefully handle self-assignment.
- C++ Core Guidelines [11]: C.62: Make copy assignment safe for self-assignment.

Rule A12-8-6 (required, implementation, automated)
Copy and move constructors and copy assignment and move assignment operators shall be declared protected or defined “=delete” in base class.

Rationale

Invoking copy or move constructor or copy assignment or move assignment operator from the top of a class hierarchy bypasses the underlying implementations. This results in “slicing” where only the base sub-objects being copied or moved.

Example

```
1  // $Id: A12-8-6.cpp 289436 2017-10-04 10:45:23Z michal.szczepankiewicz $
2  #include <memory>
3  #include <utility>
4  #include <vector>
5  class A  // Abstract base class
6  {
7    public:
8      A() = default;
9      A(A const&) = default;  // Non-compliant
10     A(A&&) = default;       // Non-compliant
11     virtual ~A() = 0;
12     A& operator=(A const&) = default;  // Non-compliant
```

```
13      A& operator=(A&&) = default;        // Non-compliant
14  };
15  class B : public A
16  {
17  };
18  class C  // Abstract base class
19  {
20    public:
21      C() = default;
22      virtual ~C() = 0;
23
24    protected:
25      C(C const&) = default;              // Compliant
26      C(C&&) = default;                   // Compliant
27      C& operator=(C const&) = default;   // Compliant
28      C& operator=(C&&) = default;        // Compliant
29  };
30  class D : public C
31  {
32  };
33  class E  // Abstract base class
34  {
35    public:
36      E() = default;
37      virtual ~E() = 0;
38      E(E const&) = delete;               // Compliant
39      E(E&&) = delete;                    // Compliant
40      E& operator=(E const&) = delete;    // Compliant
41      E& operator=(E&&) = delete;         // Compliant
42  };
43  class F : public E
44  {
45  };
46  class G  // Non-abstract base class
47  {
48    public:
49      G() = default;
50      virtual ~G() = default;
51      G(G const&) = default;              // Non-compliant
52      G(G&&) = default;                   // Non-compliant
53      G& operator=(G const&) = default;   // Non-compliant
54      G& operator=(G&&) = default;        // Non-compliant
55  };
56  class H : public G
57  {
58  };
59  void Fn1() noexcept
60  {
61      B obj1;
62      B obj2;
63      A* ptr1 = &obj1;
```

```
64      A* ptr2 = &obj2;
65      *ptr1 = *ptr2;              // Partial assignment only
66      *ptr1 = std::move(*ptr2);   // Partial move only
67      D obj3;
68      D obj4;
69      C* ptr3 = &obj3;
70      C* ptr4 = &obj4;
71      //*ptr3 = *ptr4;  // Compilation error - copy assignment operator of class C
72      // is protected
73      //*ptr3 = std::move(*ptr4);  // Compilation error - move assignment operator
74      // of class C is protected
75      F obj5;
76      F obj6;
77      E* ptr5 = &obj5;
78      E* ptr6 = &obj6;
79      //*ptr5 = *ptr6;  // Compilation error - use of deleted copy assignment
80      // operator
81      //*ptr5 = std::move(*ptr6);  // Compilation error - use of deleted move
82      // assignment operator
83      H obj7;
84      H obj8;
85      G* ptr7 = &obj7;
86      G* ptr8 = &obj8;
87      *ptr7 = *ptr8;              // Partial assignment only
88      *ptr7 = std::move(*ptr8);   // Partial move only
89  }
90  class I  // Non-abstract base class
91  {
92    public:
93      I() = default;
94      ~I() = default;
95
96    protected:
97      I(I const&) = default;              // Compliant
98      I(I&&) = default;                   // Compliant
99      I& operator=(I const&) = default;   // Compliant
100     I& operator=(I&&) = default;        // Compliant
101 };
102 class J : public I
103 {
104   public:
105     J() = default;
106     ~J() = default;
107     J(J const&) = default;
108     J(J&&) = default;
109     J& operator=(J const&) = default;
110     J& operator=(J&&) = default;
111 };
112 void Fn2() noexcept
113 {
114     std::vector<I> v1;
```

```
115     // v1.push_back(J{});  // Compilation-error on calling a deleted move
116     // constructor of I class, slicing does not occur
117     // v1.push_back(I{});  // Compilation-error on calling a deleted move
118     // constructor of I class
119
120     std::vector<J> v2;
121     v2.push_back(J{});  // No compilation error
122
123     std::vector<std::unique_ptr<I>> v3;
124     v3.push_back(std::unique_ptr<I>{});   // No compilation error
125     v3.push_back(std::make_unique<I>());  // No compilation error
126     v3.push_back(std::make_unique<J>());  // No compilation error
127     v3.emplace_back();                    // No compilation error
128 }
```

See also

- MISRA C++ 2008 [7]: Rule 12-8-2 The copy assignment operator shall be declared protected or private in an abstract class.
- HIC++ v4.0 [9]: 12.5.8 Make the copy assignment operator of an abstract class protected or define it =delete.
- C++ Core Guidelines [11]: C.67: A base class should suppress copying, and provide a virtual clone instead if "'copying"' is desired.
- C++ Core Guidelines [11]: C.81: Use =delete when you want to disable default behavior (without wanting an alternative).

Rule A12-8-7 (advisory, implementation, automated)
Assignment operators should be declared with the ref-qualifier &.

Rationale

User declared assignment operators differ from built-in operators in a way that they accept rvalues as parameters, which is confusing. Adding & to the function declaration prohibits rvalue parameters and ensures that all of the calls can only be made on lvalue objects, which results with the same behavior as for built-in types.

Note that this rule applies to all assignment operators, e.g. operator=(), operator*=(), operator+=.

Example

```
1  // $Id: A12-8-7.cpp 289436 2017-10-04 10:45:23Z michal.szczepankiewicz $
2  #include <cstdint>
3  class A
4  {
5    public:
6      A() = default;
7      A& operator*=(std::int32_t i)  // Non-compliant
```

```
8       {
9           // ...
10          return *this;
11      }
12  };
13  A F1() noexcept
14  {
15      return A{};
16  }
17  class B
18  {
19    public:
20      B() = default;
21      B& operator*=(std::int32_t) &  // Compliant
22      {
23          // ...
24          return *this;
25      }
26  };
27  B F2() noexcept
28  {
29      return B{};
30  }
31  std::int32_t F3() noexcept
32  {
33      return 1;
34  }
35  int main(int, char**)
36  {
37      F1() *= 10;  // Temporary result of f1() multiplied by 10. No compile-time
38                   // error.
39      ;
40      // f2() *= 10;  // Compile-time error due to ref-qualifier
41      ;
42      // f3() *= 10;  // Compile-time error on built-in type
43  }
```

See also

- HIC++ v4.0 [9]: 12.5.7 Declare assignment operators with the ref-qualifier &.
- cppreference.com [16]: Assignment operators.

6.13 Overloading

6.13.1 Overloadable declarations

Rule A13-1-2 (required, implementation, automated)
User defined suffixes of the user defined literal operators shall start with underscore followed by one or more letters.

Rationale

Suffixes that do not begin with the underscore character are reserved for operators provided by the standard library.

Example

```
// $Id: A13-1-2.cpp 289436 2017-10-04 10:45:23Z michal.szczepankiewicz $
constexpr long double operator"" _m(long double meters)  // Compliant
{
    // Implementation
    return meters;
}
constexpr long double operator"" _kg(long double kilograms)  // Compliant
{
    // Implementation
    return kilograms;
}
constexpr long double operator"" m(long double meters)  // Non-compliant
{
    // Implementation
    return meters;
}
constexpr long double operator"" kilograms(
    long double kilograms)  // Non-compliant
{
    // Implementation
    return kilograms;
}
void Fn()
{
    long double weight = 20.0_kg;
    long double distance = 204.8_m;
}
```

See also

- none

Rule A13-1-3 (required, implementation, automated)
User defined literals operators shall only perform conversion of passed parameters.

Rationale

It is expected behavior that the user-defined literals operators are only used to convert passed parameters to the type of declared return value. User-defined literals are not supposed to provide any other side-effects.

Example

```
// $Id: A13-1-3.cpp 289436 2017-10-04 10:45:23Z michal.szczepankiewicz $
#include <cstdint>
#include <iostream>
struct Cube
{
    unsigned long long int volume;
    constexpr explicit Cube(unsigned long long int v) : volume(v) {}
};
constexpr Cube operator"" _m3(unsigned long long int volume)
{
    return Cube(volume);  // Compliant
}
struct Temperature
{
    unsigned long long int kelvins;
    constexpr explicit Temperature(unsigned long long int k) : kelvins(k) {}
};
constexpr Temperature operator"" _K(unsigned long long int kelvins)
{
    return Temperature(kelvins);  // Compliant
}
static void SumDistances(std::int32_t distance)
{
    static std::int32_t overallDistance = 0;
    overallDistance += distance;
}
struct Distance
{
    long double kilometers;
    explicit Distance(long double kms) : kilometers(kms) {}
};
Distance operator"" _m(long double meters)
{
    SumDistances(meters);  // Non-compliant - function has a side-effect
    return Distance(meters / 1000);
}
void operator"" _print(const char* str)
{
    std::cout << str << '\n';  // Non-compliant - user-defined literal operator
                               // does not perform conversion and has a
                               // side-effect
}
```

See also

- none

6.13.2 Declaration matching

Rule A13-2-1 (required, implementation, automated)
An assignment operator shall return a reference to "this".

Rationale

Returning a type "T&" from an assignment operator is consistent with the C++ Standard Library.

Example

```
// $Id: A13-2-1.cpp 271687 2017-03-23 08:57:35Z piotr.tanski $
class A
{
  public:
    // ...
    A& operator=(const A&) &  // Compliant
    {
        // ...
        return *this;
    }
};

class B
{
  public:
    // ...
    const B& operator=(const B&) &  // Non-compliant - violating consistency
                                    // with standard types
    {
        // ...
        return *this;
    }
};

class C
{
  public:
    // ...
    C operator=(const C&) &  // Non-compliant
    {
        // ...
        return *this;
    }
};

```

```
36 class D
37 {
38   public:
39     // ...
40     D* operator=(const D&) &  // Non-compliant
41     {
42         // ...
43         return this;
44     }
45 };
```

See also

- HIC++ v4.0 [9]: 13.2.2 Ensure that the return type of an overloaded binary operator matches the built-in counterparts.
- C++ Core Guidelines [11]: F.47: Return T& from assignment operators.

Rule A13-2-2 (required, implementation, automated)
A binary arithmetic operator and a bitwise operator shall return a "prvalue".

Rationale

Returning a type "T" from binary arithmetic and bitwise operators is consistent with the C++ Standard Library.

See: prvalue.

Example

```
1  // $Id: A13-2-2.cpp 271687 2017-03-23 08:57:35Z piotr.tanski $
2  #include <cstdint>
3
4  class A
5  {
6  };
7
8  A operator+(A const&, A const&) noexcept  // Compliant
9  {
10     return A{};
11 }
12 std::int32_t operator/(A const&, A const&) noexcept  // Compliant
13 {
14     return 0;
15 }
16 A operator&(A const&, A const&)noexcept  // Compliant
17 {
18     return A{};
19 }
20 const A operator-(A const&, std::int32_t) noexcept  // Non-compliant
21 {
```

```
22      return A{};
23  }
24  A* operator|(A const&, A const&) noexcept  // Non-compliant
25  {
26      return new A{};
27  }
```

See also

- HIC++ v4.0 [9]: 13.2.2 Ensure that the return type of an overloaded binary operator matches the built-in counterparts.

Rule A13-2-3 (required, implementation, automated)
A relational operator shall return a boolean value.

Rationale

Returning a type “bool” from a relational operator is consistent with the C++ Standard Library.

Example

```
1   // $Id: A13-2-3.cpp 271687 2017-03-23 08:57:35Z piotr.tanski $
2   #include <cstdint>
3
4   class A
5   {
6   };
7
8   bool operator==(A const&, A const&) noexcept  // Compliant
9   {
10      return true;
11  }
12  bool operator<(A const&, A const&) noexcept  // Compliant
13  {
14      return false;
15  }
16  bool operator!=(A const& lhs, A const& rhs) noexcept  // Compliant
17  {
18      return !(operator==(lhs, rhs));
19  }
20  std::int32_t operator>(A const&, A const&) noexcept  // Non-compliant
21  {
22      return -1;
23  }
24  A operator>=(A const&, A const&) noexcept  // Non-compliant
25  {
26      return A{};
27  }
28  const A& operator<=(A const& lhs, A const& rhs) noexcept  // Non-compliant
```

```
{
    return lhs;
}
```

See also

- HIC++ v4.0 [9]: 13.2.2 Ensure that the return type of an overloaded binary operator matches the built-in counterparts.

6.13.3 Overload resolution

Rule A13-3-1 (required, implementation, automated)
A function that contains "forwarding reference" as its argument shall not be overloaded.

Rationale

A template parameter that is declared "T&&" (Scott Meters called it a "universal reference", while C++ Language Standard calls it a "forwarding reference") will deduce for any type. Overloading functions with "forwarding reference" argument may lead to developer's confusion on which function will be called.

Exception

Declaring an overloading function that takes a "forwarding reference" parameter to be "=delete" does not violate this rule.

Declaring a "forwarding constructor" that is constrained (via SFINAE) to not match any other overloads also does not violate this rule, see A14-5-1.

Example

```
// $Id: A13-3-1.cpp 309903 2018-03-02 12:54:18Z christof.meerwald $
#include <cstdint>
template <typename T>
void F1(T&& t) noexcept(false)
{
}
void F1(
    std::int32_t&& t) noexcept  // Non-compliant - overloading a function with
                                // forwarding reference
{
}
template <typename T>
void F2(T&& t) noexcept(false)
{
}
void F2(std::int32_t&) = delete;  // Compliant by exception

```

```
18 class A
19 {
20   public:
21     // Compliant by exception, constrained to not match copy/move ctors
22     template<typename T,
23              std::enable_if_t<! std::is_same<std::remove_cv_t<std::
     remove_reference_t<T>>, A>::value> * = nullptr>
24     A(T &&value);
25 };
26
27 int main(int, char**)
28 {
29     std::int32_t x = 0;
30     F1(x);   // Calls f1(T&&) with T = int&
31     F1(+x);  // Calls f1(std::int32_t&&)
32     F1(0);   // Calls f1(std::int32_t&&)
33     F1(0U);  // Calls f1(T&&) with T = unsigned int
34     F2(0);   // Calls f2(T&&) with T = int
35     // f2(x);  // Compilation error, the overloading function is deleted
36 }
```

See also

- HIC++ v4.0 [9]: 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.
- Effective Modern C++ [13]: Item 26. Avoid overloading on universal references.

6.13.5 Overloaded operators

Rule A13-5-1 (required, implementation, automated)
If "operator[]" is to be overloaded with a non-const version, const version shall also be implemented.

Rationale

A non-const overload of the subscript operator allows an object to be modified, by returning a reference to member data, but it does not allow reading from const objects. The const version of "operator[]" needs to be implemented to ensure that the operator can be invoked on a const object.

Note that one can provide a const version of operator[] (to support read-only access to elements), but without a non-const version.

Example

```
1 // $Id: A13-5-1.cpp 289436 2017-10-04 10:45:23Z michal.szczepankiewicz $
2 #include <cstdint>
```

```
class Container1
{
  public:
    std::int32_t& operator[](
        std::int32_t index)  // Compliant - non-const version
    {
        return container[index];
    }
    std::int32_t operator[](
        std::int32_t index) const  // Compliant - const version
    {
        return container[index];
    }

  private:
    static constexpr std::int32_t maxSize = 10;
    std::int32_t container[maxSize];
};
void Fn() noexcept
{
    Container1 c1;
    std::int32_t e = c1[0];  // Non-const version called
    c1[0] = 20;              // Non-const version called
    Container1 const c2{};
    e = c2[0];  // Const version called
    // c2[0] = 20;  // Compilation error
}
class Container2  // Non-compliant - only non-const version of operator[]
                  // implemented
{
  public:
    std::int32_t& operator[](std::int32_t index) { return container[index]; }

  private:
    static constexpr std::int32_t maxSize = 10;
    std::int32_t container[maxSize];
};
```

See also

- HIC++ v4.0 [9]: 13.2.4 When overloading the subscript operator (operator[]) implement both const and non-const versions.

Rule A13-5-2 (required, implementation, automated)
All user-defined conversion operators shall be defined explicit.

Rationale

Without explicit keyword, a single user defined conversion can be invoked in a standard conversion sequence, which can lead to accidental errors.

Example

```
// $Id: A13-5-2.cpp 303121 2018-01-09 09:03:52Z michal.szczepankiewicz $
class A
{
  public:
    explicit A(double d) : d(d) {}
    explicit operator double() const { return d; }  // Compliant
  private:
    double d;
};

int main(void)
{
    A a{3.14159265358979323846264338327950288419716939937510582097494459230 78};

    double tmp1{a};
    // float tmp2{a}; //compilation error instead of warning, prevents from data
    // precision loss

    return 0;
}
```

See also

- HIC++ v4.0 [9]: 12.1.1: Do not declare implicit user defined conversions.

Rule A13-5-3 (advisory, implementation, automated)
User-defined conversion operators should not be used.

Rationale

Explicitly named conversions using dedicated member function eliminate any potential errors that can arise if the type conversion operators have to be used.

If using conversion operators is fundamental in an application domain, see A13-5-2.

Example

```
// $Id: A13-5-3.cpp 305629 2018-01-29 13:29:25Z piotr.serwa $
#include <iostream>

class Complex
{
public:
    Complex (double r, double i = 0.0) : re(r), im(i) {}
    explicit operator double() const noexcept { return re; }
    double AsDouble() const noexcept { return re; }
private:
    double re;
    double im;
```

```
};

int main(void)
{
   Complex c(2.0f);

   std::cout << (double) c << std::endl;   //compliant with A13-5-2, non-compliant with A13-5-3
   std::cout << c.AsDouble() << std::endl; //compliant

   return 0;
}
```

See also

- JSF December 2005 [8]: AV Rule 177: User-defined conversion functions should be avoided.
- C++ Core Guidelines [11]: C.164: Avoid conversion operators.

Rule A13-5-4 (required, implementation, automated)
If two opposite operators are defined, one shall be defined in terms of the other.

Rationale

Defining one operator in terms of the other simplifies maintenance and prevents from accidental errors during code development.

Note: Completeness of relational operators can be achieved by implementing just `operator==` and `operator<` and using namespace `rel_ops`.

Example

```
// $Id: A13-5-4.cpp 328319 2018-08-03 14:08:42Z christof.meerwald $
#include <cstdint>

// non-compliant
class A
{
public:
    explicit A(std::uint32_t d) : d(d) {}

    friend bool operator==(A const & lhs, A const & rhs) noexcept
    {
        return lhs.d == rhs.d;
    }
    friend bool operator!=(A const & lhs, A const & rhs) noexcept
    {
        return lhs.d != rhs.d;
    }
```

```
18
19  private:
20      std::uint32_t d;
21  };
22
23  // compliant
24  class B
25  {
26  public:
27      explicit B(std::uint32_t d) : d(d) {}
28
29      friend bool operator==(B const & lhs, B const & rhs) noexcept
30      {
31          return lhs.d == rhs.d;
32      }
33
34      friend bool operator!=(B const & lhs, B const & rhs) noexcept
35      {
36          return !(lhs == rhs);
37      }
38  private:
39      std::uint32_t d;
40  };
```

See also

- JSF December 2005 [8]: AV Rule 85: When two operators are opposites (such as == and !=), both will be defined and one will be defined in terms of the other.

Rule A13-5-5 (required, implementation, automated)
Comparison operators shall be non-member functions with identical parameter types and noexcept.

Rationale

Any asymmetric behavior for comparison functions can be confusing. In order to achieve fully symmetric treatment, comparison functions need to be defined as non-member functions, as the implicit object parameter of a member function does not allow user-defined conversions to be applied (but the right hand side would).

Since comparison is a fundamental operation, it should never throw an exception.

Note: This rule applies to ==, !=, <, <=, >, and >=

Note: Declaring a comparison operator as a friend allows it to access internal data similar to a member function and is allowed by exception in rule A11-3-1.

Example

```
1  // $Id: A13-5-5.cpp 325916 2018-07-13 12:26:22Z christof.meerwald $
2  #include <cstdint>
```

```
3
4  class A
5  {
6  public:
7      explicit A(std::uint32_t d)
8        : m_d(d)
9      { }
10
11     bool operator ==(A const & rhs) const // Non-compliant: member, not noexcept
12     {
13         return m_d == rhs.m_d;
14     }
15
16 private:
17     std::uint32_t m_d;
18 };
19
20 class C
21 {
22 public:
23     operator A() const;
24 };
25
26 void Foo(A const & a, C const & c)
27 {
28     a == c; // asymmetric as "a ==c" compiles, but "c == a" doesn't compile
29 }
30
31
32 class B
33 {
34 public:
35     explicit B(std::uint32_t d)
36       : m_d(d)
37     { }
38
39     // Compliant: non-member, identical parameter types, noexcept
40     friend bool operator ==(B const & lhs, B const & rhs) noexcept
41     {
42         return lhs.m_d == rhs.m_d;
43     }
44
45 private:
46     std::uint32_t m_d;
47 };
48
49 class D
50 {
51 public:
52     operator B() const;
53 };
```

```
54
55 void Bar(B const & b, D const & d)
56 {
57     b == d;
58     d == b;
59 }
```

See also

- C++ Core Guidelines [11]: C.86: Make == symmetric with respect to operand types and noexcept

6.13.6 Build-in operators

Rule A13-6-1 (required, implementation, automated)
Digit sequences separators ’ shall only be used as follows: (1) for decimal, every 3 digits, (2) for hexadecimal, every 2 digits, (3) for binary, every 4 digits.

Rationale

Since C++14 Language Standard it is allowed (optionally) to separate any two digits in digit sequences with separator ’. However, to meet developer expectations, usage of separator in integer and floating-point digit sequences should be unified:

- for decimal values, separator can be placed every 3 digits, e.g. 3’000’000, 3.141’592’653
- for hexadecimal values, separator can be placed every 2 digits, e.g. 0xFF’FF’FF’FF
- for binary values, separator can be placed very 4 digits, e.g. 0b1001’1101’0010

Example

```
1  // $Id: A13-6-1.cpp 289436 2017-10-04 10:45:23Z michal.szczepankiewicz $
2  #include <cstdint>
3  void Fn() noexcept
4  {
5      std::uint32_t decimal1 = 3’000’000;    // Compliant
6      std::uint32_t decimal2 = 4’500;        // Compliant
7      std::uint32_t decimal3 = 54’00’30;     // Non-compliant
8      float decimal4 = 3.141’592’653;        // Compliant
9      float decimal5 = 3.1’4159’265’3;       // Non-compliant
10     std::uint32_t hex1 = 0xFF’FF’FF’FF;    // Compliant
11     std::uint32_t hex2 = 0xFAB’1’FFFFF;    // Non-compliant
12     std::uint8_t binary1 = 0b1001’0011;    // Compliant
13     std::uint8_t binary2 = 0b10’00’10’01;  // Non-compliant
14 }
```

See also

- ISO 26262-6 [5]: 8.4.4 e) readability and comprehensibility

6.14 Templates

6.14.0 General

6.14.1 Template parameters

Rule A14-1-1 (advisory, implementation, non-automated)
A template should check if a specific template argument is suitable for this template.

Rationale

If a template class or function requires specific characteristics from a template type (e.g. if it is move constructible, copyable, etc.), then it needs to check whether the type matches the requirements to detect possible faults. The goal of this rule is to ensure that a template defines all of the preconditions that a template argument needs to fulfill without having any information about the specific class.

This can be achieved in compile time using static_assert assertion.

Example

```
1  // $Id: A14-1-1.cpp 289436 2017-10-04 10:45:23Z michal.szczepankiewicz $
2  #include <utility>
3  class A
4  {
5    public:
6      A() = default;
7      ~A() = default;
8      A(A const&) = delete;
9      A& operator=(A const&) = delete;
10     A(A&&) = delete;
11     A& operator=(A&&) = delete;
12 };
13 class B
14 {
15   public:
16     B() = default;
17     B(B const&) = default;
18     B& operator=(B const&) = default;
19     B(B&&) = default;
20     B& operator=(B&&) = default;
21 };
22 template <typename T>
```

```
void F1(T const& obj) noexcept(false)
{
    static_assert(
        std::is_copy_constructible<T>(),
        "Given template type is not copy constructible.");  // Compliant
}
template <typename T>
class C
{
    // Compliant
    static_assert(std::is_trivially_copy_constructible<T>(),
                  "Given template type is not trivially copy constructible.");

    // Compliant
    static_assert(std::is_trivially_move_constructible<T>(),
                  "Given template type is not trivially move constructible.");

    // Compliant
    static_assert(std::is_trivially_copy_assignable<T>(),
                  "Given template type is not trivially copy assignable.");

    // Compliant
    static_assert(std::is_trivially_move_assignable<T>(),
                  "Given template type is not trivially move assignable.");

  public:
    C() = default;
    C(C const&) = default;
    C& operator=(C const&) = default;
    C(C&&) = default;
    C& operator=(C&&) = default;

  private:
    T c;
};
template <typename T>
class D
{
  public:
    D() = default;
    D(D const&) = default;              // Non-compliant - T may not be copyable
    D& operator=(D const&) = default;   // Non-compliant - T may not be copyable
    D(D&&) = default;                   // Non-compliant - T may not be movable
    D& operator=(D&&) = default;        // Non-compliant - T may not be movable

  private:
    T d;
};
void F2() noexcept
{
    A a;
```

```
74      B b;
75      // f1<A>(a);  // Class A is not copy constructible, compile-time error
76      // occurs
77      F1<B>(b);  // Class B is copy constructible
78      // C<A> c1;  // Class A can not be used for template class C, compile-time
79      // error occurs
80      C<B> c2;  // Class B can be used for template class C
81      D<A> d1;
82      // D<A> d2 = d1;  // Class D can not be copied, because class A is not
83      // copyable, compile=time error occurs
84      // D<A> d3 = std::move(d1);  // Class D can not be moved, because class A is
85      // not movable, compile-time error occurs
86      D<B> d4;
87      D<B> d5 = d4;
88      D<B> d6 = std::move(d4);
89  }
```

See also

- JSF December 2005 [8]: AV Rule 103: Constraint checks should be applied to template arguments.
- C++ Core Guidelines [11]: T.150: Check that a class matches a concept using static_assert.

6.14.5 Template declarations

Rule A14-5-1 (required, implementation, automated)
A template constructor shall not participate in overload resolution for a single argument of the enclosing class type.

Rationale

A template constructor is never a copy or move constructor and therefore doesn't prevent the implicit definition of a copy or move constructor even if the template constructor looks similar and might easily be confused.

At the same time, copy or move operations do not necessarily only use a copy or move constructor, but go through the normal overload resolution process to find the best matching function to use.

This can cause confusion in the following cases:

- a template constructor that looks like a copy/move constructor is not selected for a copy/move operation because the compiler has generated an implicit copy/move constructor as well
- a template constructor is selected in preference over a copy/move constructor because the template constructor is a better match

To avoid these confusing situations, template constructors shall not participate in overload resolution for a single argument of the enclosing class type to avoid a template constructor being selected for a copy/move operation. It also makes it clear that the constructor is not a copy/move constructor and that it does not prevent the implicit generation of copy/move constructors.

Example

```
// $Id: A14-5-1.cpp 309903 2018-03-02 12:54:18Z christof.meerwald $
#include <cstdint>
#include <type_traits>

class A
{
  public:
    // Compliant: template constructor does not participate in overload
    //            resolution for copy/move operations
    template<typename T,
             std::enable_if_t<! std::is_same<std::remove_cv_t<T>, A>::value> * =
    nullptr>
    A(const T &value)
      : m_value { value }
    { }

  private:
    std::int32_t m_value;
};

void Foo(A const &a)
{
    A myA { a }; // will use the implicit copy ctor, not the template converting
    ctor

    A a2 { 2 };  // will use the template converting ctor
}

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

    // Compliant: forwarding constructor does not participate in overload
    //            resolution for copy/move operations
    template<typename T,
             std::enable_if_t<! std::is_same<std::remove_cv_t<std::
    remove_reference_t<T>>, B>::value> * = nullptr>
    B(T &&value);
};

void Bar(B b)
{
```

```
    B myB { b };  // will use the copy ctor, not the forwarding ctor
}

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

    // Non-Compliant: unconstrained template constructor
    template<typename T>
    C(T &);
};

void Bar(C c)
{
    C myC { c };  // will use template ctor instead of copy ctor
}
```

See also

- MISRA C++ 2008 [7]: M14-5-2: A copy constructor shall be declared when there is a template constructor with a single parameter that is a generic parameter.

Rule A14-5-2 (advisory, design, partially-automated)
Class members that are not dependent on template class parameters should be defined in a separate base class.

Rationale

Having non-dependent members in a class template can lead to unnecessary template instantiations and potential code bloat. It is therefore preferred to move those members to a non-dependent base class so they can be used without any template instantiation.

Example

```
// $Id: A14-5-2.cpp 323444 2018-06-22 14:38:18Z christof.meerwald $
#include <cstdint>

template<typename T>
class A
{
  public:
    enum State  // Non-Compliant: member doesn't depend on template parameter
    {
        State1,
        State2
    };

    State GetState();
```

```
};

class B_Base
{
  public:
    enum State  // Compliant: not a member of a class template
    {
        State1,
        State2
    };
};

template<typename T>
class B : B_Base
{
  public:
    State GetState();
};
```

See also

- C++ Core Guidelines [11]: T.62: Place non-dependent class template members in a non-templated base class

Rule A14-5-3 (advisory, design, automated)
A non-member generic operator shall only be declared in a namespace that does not contain class (struct) type, enum type or union type declarations.

Rationale

Argument-dependent lookup (ADL) adds additional associated namespaces to the set of scopes searched when lookup is performed for the names of called functions. A generic operator found in one of these additional namespaces would be added to the overload set and choosen by overload resolution. ADL is complicated by several possible use forms for operators (via function calls and via expression, operators can be declared as members and as non-members) and lookup in those cases is different, which is likely to be inconsistent with developer expectation.

Generic operator is a non-member operator template that can be called without explicit template arguments and has at least one generic parameter. A template type parameter T is a generic parameter if, in the function declaration, it has the (possibly cv-qualified) form of T, or T & or T &&.

Example

```
// $Id: A14-5-3.cpp $
#include <cstdint>

template<typename T>
```

```
5  class B
6  {
7    public:
8      bool operator+( long rhs );
9
10     void f()
11     {
12         *this + 10;
13     }
14 };
15
16 namespace NS1
17 {
18     class A {};
19
20     template<typename T>
21     bool operator+( T, std::int32_t ); // Non-Compliant: a member of namespace
22                                        // with other declarations
23 }
24
25 namespace NS2
26 {
27     void g();
28
29     template<typename T>
30     bool operator+( T, std::int32_t ); // Compliant: a member of namespace
31                                        // with declarations of functions only
32 }
33
34 template class B<NS1::A>; // NS1::operator+ will be used in function B::f()
35                           // instead of B::operator+
```

See also

- MISRA C++ 2008 [7]: M14-5-1: A non-member generic function shall only be declared in a namespace that containes only operator declarations.

Rule M14-5-3 (required, implementation, automated)
A copy assignment operator shall be declared when there is a template assignment operator with a parameter that is a generic parameter.

See MISRA C++ 2008 [7]

6.14.6 Name resolution

Rule M14-6-1 (required, implementation, automated)
In a class template with a dependent base, any name that may be found in that dependent base shall be referred to using a qualified-id or this->.

See MISRA C++ 2008 [7]

6.14.7 Template instantiation and specialization

Rule A14-7-1 (required, implementation, automated)
A type used as a template argument shall provide all members that are used by the template.

Rationale

If a type used as a template argument does not provide all the members used by the template, the instantiation of the template will result in an ill-formed program. It is not clear for developer whether the template should be used with the type.

Example

```
1  // $Id: A14-7-1.cpp 289436 2017-10-04 10:45:23Z michal.szczepankiewicz $
2  #include <cstdint>
3  class A
4  {
5    public:
6      void SetProperty(std::int32_t x) noexcept { property = x; }
7      void DoSomething() noexcept {}
8
9    private:
10     std::int32_t property;
11 };
12 struct B
13 {
14 };
15 class C
16 {
17   public:
18     void DoSomething() noexcept {}
19 };
20 template <typename T>
21 class D
22 {
23   public:
24     void F1() {}
25     void F2()
26     {
27         T t;
```

```
28          t.SetProperty(0);
29      }
30      void F3()
31      {
32          T t;
33          t.DoSomething();
34      }
35  };
36
37  void Fn() noexcept
38  {
39      D<A> d1;  // Compliant - struct A provides all needed members
40      d1.F1();
41      d1.F2();
42      d1.F3();
43
44      D<B> d2;  // Non-compliant - struct B does not provide needed members
45      d2.F1();
46      // d2.f2();  // Compilation error - no 'property' in struct B
47      // d2.f3();  // Compilation error - no member named 'doSomething' in struct
48      // B
49
50      D<C> d3;  // Non-compliant - struct C does not provide property
51      d3.F1();
52      // d3.F2();  // Compilation error - no property in struct C
53      d3.F3();
54  }
```

See also

- MISRA C++ 2008 [7]: Rule 14-7-2 (Required) For any given template specialization, an explicit instantiation of the template with the template arguments used in the specialization shall not render the program ill-formed.

Rule A14-7-2 (required, implementation, automated)
Template specialization shall be declared in the same file (1) as the primary template (2) as a user-defined type, for which the specialization is declared.

Rationale

It is undefined behavior, when a compiler sees the (partial or explicit) template specialization after it already has instantiated the primary or less specialized template. Moreover, the case (2) allows compile-time interfaces to be extensible, as developers can safely provide custom specializations e.g. for traits classes or `std::hash`.

Example

```
1  // $Id: A14-7-2.cpp 312645 2018-03-21 11:44:35Z michal.szczepankiewicz $
2  #include <cstdint>
3
```

```
//in A.hpp

#include <functional>

struct A
{
    std::uint8_t x;
};

namespace std {

//compliant, case (2)
//template specialization for the user-defined type
//in the same file as the type declaration
template <>
struct hash<A>
{
    size_t operator()(const A& a) const noexcept
    {
        return std::hash<decltype(a.x)>()(a.x);
    }
};

}

//traits.hpp

#include <type_traits>
#include <cstdint>

template <typename T>
struct is_serializable : std::false_type {};

//compliant, case (1)
template <>
struct is_serializable<std::uint8_t> : std::true_type {};

//func.cpp

#include <vector>

//non-compliant, not declared
//in the same file as
//is_serializable class
template <>
struct is_serializable<std::uint16_t> : std::true_type {};

template <typename T, typename = std::enable_if<is_serializable<T>::value>>
std::vector<std::uint8_t> serialize(const T& t)
{
    //only a basic stub
```

```
55      return std::vector<std::uint8_t>{t};
56  }
57
58  #include <string>
59  int main()
60  {
61      serialize(std::uint8_t{3});
62  }
```

See also

- MISRA C++ 2008 [7]: Rule 14-7-2 (Required) For any given template specialization, an explicit instantiation of the template with the template arguments used in the specialization shall not render the program ill-formed.

6.14.8 Function template specializations

Rule A14-8-2 (required, implementation, automated)
Explicit specializations of function templates shall not be used.

Rationale

Specializations of function templates do not participate in overload resolution. They are only considered after their primary template has been selected during overload resolution. This is highly dependent on the declaration order of overloads and specializations and may be inconsistent with developer expectations.

A non-template function is always selected over a function template specialization if they are otherwise an equally good match, which also may be confusing for developers.

Function templates cannot be partially specialized, which may lead to troublesome implementations. If a partial specialization is required, then it is recommended to write a single function template that delegates to a class template (which can be partially specialized).

Example

```
1  // $Id: A14-8-2.cpp 312698 2018-03-21 13:17:36Z michal.szczepankiewicz $
2  #include <cstdint>
3  #include <memory>
4  #include <iostream>
5
6  template <typename T>
7  void F1(T t)
8  {
9      //compliant, (a)
10     std::cout << "(a)" << std::endl;
11 }
```

```
12
13  template <>
14  void F1<>(uint16_t* p)
15  {
16      //non-compliant
17      //(x), explicit specialization of
18      //(a), not (b), due to declaration
19      //order
20      std::cout << "(x)" << std::endl;
21  }
22
23  template <typename T>
24  void F1(T* p)
25  {
26      //compliant, (b), overloads (a)
27      std::cout << "(b)" << std::endl;
28  }
29
30  template <>
31  void F1<>(uint8_t* p)
32  {
33      //non-compliant
34      //(c), explicit specialization of (b)
35      std::cout << "(c)" << std::endl;
36  }
37
38  void F1(uint8_t* p)
39  {
40      //compliant
41      //(d), plain function, overloads with (a), (b)
42      //but not with (c)
43      std::cout << "(d)" << std::endl;
44  }
45
46  int main(void)
47  {
48      auto sp8 = std::make_unique<uint8_t>(3);
49      auto sp16 = std::make_unique<uint16_t>(3);
50
51      F1(sp8.get()); //calls (d), which might be
52                     //confusing, but (c) is non-compliant
53
54      F1(sp16.get()); //calls (b), which might be
55                      //confusing, but (b) is non-compliant
56  }
```

See also

- MISRA C++ 2008 [7]: 14-8-1: Overloaded function templates shall not be explicitly specialized.

- MISRA C++ 2008 [7]: 14-8-2: The viable function set for a function call should either contain no function specializations, or only contain function specializations.
- HIC++ v4.0 [9]: 14.2.2: Do not explicitly specialize a function template that is overloaded with other templates.
- C++ Core Guidelines [11]: T.144: Don't specialize function templates.

6.15 Exception handling

Advantages of using exceptions

"The exception handling mechanism can provide an effective and clear means of handling error conditions, particularly where a function needs to return both some desired result together with an indication of success or failure. However, because of its ability to transfer control back up the call tree, it can also lead to code that is difficult to understand. Hence it is required that the mechanism is only used to capture behavior that is in some sense undesirable, and which is not expected to be seen in normal program execution." [MISRA C++ 2008]

"The preferred mechanism for reporting errors in a C++ program is exceptions rather than using error codes. A number of core language facilities, including dynamic_cast, operator new(), and typeid, report failures by throwing exceptions. In addition, the C++ standard library makes heavy use of exceptions to report several different kinds of failures. Few C++ programs manage to avoid using some of these facilities." [ISO C++ Core Guidelines].

Consequently, C++ programs need to be prepared for exceptions to occur and need to handle each appropriately.

Challenges of using exceptions

Issue:	**Solution:**
Correctness of the exception handling	Exception handling mechanism is implemented by the compiler (by its library functions and machine code generator) and defined by the C++ Language Standard. Rule A1-2-1 requires that the compiler (including its exception handling routines), when used for safety-related software, meets appropriate safety requirements.
Hidden control flow	ISO 26262-6 (Table *) recommends "no hidden data flow or control flow" for ASIL A software and highly recommends it for ASIL B/C/D. Therefore, the Rule A15-0-1 prohibits the usage of exceptions for normal control flow of software - they are allowed only for errors where a function failed to perform its assigned task.

Additional exit point from functions	ISO 26262-6 (Table *) highly recommends “one entry and one exit point in subprograms and functions” for ASIL A software. Therefore, the Rule A15-0-1 prohibits the usage of exceptions for normal control flow of software - they are allowed only for errors where a function failed to perform its assigned task.
Code readability	If exceptions are used correctly, in particularly by using checked and unchecked exception types, see Rules: A15-0-4 and A15-0-5, the code is easier to read and maintain than if using error codes. It avoids nesting if/else error-checking statements.
Exception safety and program state consistency after exception is thrown	The Rule A15-0-2 requires that functions provide at least “basic exception safety” (Note: this C++ term is not related to functional safety)
Impact on runtime performance	If a function does not throw an exception (i.e. error conditions do not occur), then there could be a little overhead due to exception handling mechanism initialization. However, some compilers offer “zero cost exception handling”, which means that there is no performance overhead if the exception is not thrown.
Impact on worst-case execution time	The A15-0-7 rule requires that the exception handling mechanism provides real-time implementation. Note that this is not the case for e.g. GCC compiler that allocates dynamic memory on throwing an exception. However, it is possible to fix it simply by avoiding memory allocation.
Maturity of exceptions	Exceptions are a widespread concept in several programming languages, not only in C++, but also in e.g. Ada, Java, Modula-3, ML, OCaml, Python, Ruby, C#, Lisp, Eiffel, and Modula-2.
Tool support	There are several tools that support exceptions well: compilers (e.g. gcc, clang, visual studio), IDEs (e.g. eclipse, clion, visual studio), static analysis tools (e.g. QA C++, Coverity Prevent) and compiler validation suites (e.g. SuperTest).
Appropriate usage of exceptions in implementation	Exceptions need to be used properly in the code, therefore this document specifies almost 40 precise rules defining how to code using exceptions, in particular defining the rules for checked/unchecked exceptions.

Table 6.1: Challenges of exceptions usage

Checked and unchecked exceptions

Like MISRA introduces a concept of "underlying type", AUTOSAR C++14 Guidelines introduces a concept of unchecked and checked exceptions. This is based on the classification used in Java language, having as a goal an efficient, complete and consistent way of specifying the exceptions. There are therefore two exclusive categories of exceptions:

- Checked Exceptions: Used to represent errors that are expected and reasonable to recover from, so they are supposed to be documented by functions using a dedicated tag (e.g. `@throws`) and have to be either handled or documented (in the same way) by caller functions. Exceptions are marked as Checked using a separate tag (e.g. `@checkedException`) that precedes an exception class declaration.
- Unchecked Exceptions: Used to represent errors that a program typically can not recover from. However, unchecked exceptions can be documented by functions, i.e in cases when all preconditions of thrown exception are defined and known. It is context dependent where such an exception can be caught (e.g. is it done before main function) and what is the proper handling (e.g. other than program termination). However, it is not forced so that unchecked exceptions are documented by caller functions (even if they are documented by called functions). By default, all exceptions are unchecked (also from third-party libraries used), unless their definition is preceded by the dedicated tag.

“Checked exceptions are a wonderful feature of the Java programming language. Unlike return codes, they force the programmer to deal with exceptional conditions, greatly enhancing reliability.” [Effective Java 2nd Edition [15]]

The following sections specify several specific rules defining the usage of exceptions, in particular concerning the use of unchecked and checked exceptions.

6.15.0 General

Rule A15-0-1 (required, architecture / design / implementation, non-automated)
A function shall not exit with an exception if it is able to complete its task.

Rationale

“The notion of an exception is provided to help get information from the point where an error is detected to a point where it can be handled. A function that cannot cope with a problem throws an exception, hoping that its (direct or indirect) caller can handle the problem. A function that wants to handle a kind of problem indicates that by catching the corresponding exception.” [The C++ Programming Language [14]]

Exceptions are only supposed to be used to capture incorrect, and which is not expected to be seen in normal program, execution. Using exception handling mechanism to transfer control back up the call stack, in error-free situation, leads to

code that is difficult to understand and significantly less efficient than returning from a function.

Note that most of the monitoring or supervision functions are not supposed to throw an exception when an error is detected.

Example

```
1  //% $Id: A15-0-1.cpp 289436 2017-10-04 10:45:23Z michal.szczepankiewicz $
2  #include <fstream>
3  #include <stdexcept>
4  #include <string>
5  #include <vector>
6  std::uint8_t ComputeCrc(std::string& msg);
7  bool IsMessageCrcCorrect1(std::string& message)
8  {
9      std::uint8_t computedCRC = ComputeCrc(message);
10     std::uint8_t receivedCRC = message.at(0);
11
12     if (computedCRC != receivedCRC)
13     {
14         throw std::logic_error(
15             "Computed CRC is invalid.");  // Non-compliant - CheckMessageCRC()
16                                           // was able to perform
17         // its task, nothing exceptional about its invalid result
18     }
19
20     return true;
21 }
22 bool IsMessageCrcCorrect2(std::string& message)
23 {
24     bool isCorrect = true;
25     std::uint8_t computedCRC = ComputeCrc(message);
26     std::uint8_t receivedCRC = message.at(0);
27
28     if (computedCRC != receivedCRC)
29     {
30         isCorrect =
31             false;  // Compliant - if CRC is not correct, then return "false"
32     }
33
34     return isCorrect;
35 }
36 void SendData(std::string message)
37 {
38     if (message.empty())
39     {
40         throw std::logic_error("Preconditions are not met.");  // Compliant -
41                                                                // SendData() was
42                                                                // not able to
43                                                                // perform its
44                                                                // task
```

```
45      }
46
47      bool sendTimeoutReached = false;
48
49      // Implementation
50      if (sendTimeoutReached)
51      {
52          throw std::runtime_error(
53              "Timeout on sending a message has been reached.");  // Compliant -
54                                                                  // SendData()
55                                                                  // did not
56                                                                  // perform its
57                                                                  // task
58      }
59  }
60  std::int32_t FindIndex(std::vector<std::int32_t>& v, std::int32_t x) noexcept
61  {
62      try
63      {
64          std::size_t size = v.size();
65          for (std::size_t i = 0U; i < size; ++i)
66          {
67              if (v.at(i) == x)  // v.at() throws an std::out_of_range exception
68              {
69                  throw i;  // Non-compliant - nothing exceptional about finding a
70                            // value in vector
71              }
72          }
73      }
74
75      catch (std::size_t
76                  foundIdx)  // Non-compliant - nothing exceptional about finding a
77                             // value in vector
78      {
79          return foundIdx;
80      }
81
82      catch (std::out_of_range&
83                  e)  // Compliant - std::out_of_range error shall be handled
84      {
85          return -1;
86      }
87
88      return -1;
89  }
90  bool ReadFile(std::string& filename) noexcept
91  {
92      try
93      {
94          std::ifstream file(filename, std::ios_base::in);
95
```

```
96          if (!file.is_open())
97          {
98              throw std::runtime_error(
99                  "File cannot be opened");  // Compliant - error on opening a
100                                            // file is an exceptional case
101         }
102
103         char c = file.get();
104
105         if (!file.good())
106         {
107             throw std::runtime_error(
108                 "Cannot read from file");  // Compliant - error on reading from
109                                            // file is an exceptional case
110         }
111     }
112
113     catch (std::exception& e)
114     {
115         return false;
116     }
117
118     return true;
119 }
120 void Fn1(
121     std::uint32_t x)  // Non-compliant - inefficient and less readable version
122                       // than its obvious alternative, e.g. fn2()
123 // function
124 {
125     try
126     {
127         if (x < 10)
128         {
129             throw 10;
130         }
131
132         // Action "A"
133     }
134
135     catch (std::int32_t y)
136     {
137         // Action "B"
138     }
139 }
140 void Fn2(
141     std::uint32_t x)  // Compliant - the same functionality as fn1() function
142 {
143     if (x < 10)
144     {
145         // Action "B"
146     }
```

```
147     else
148     {
149         // Action "A"
150     }
151 }
```

See also

- MISRA C++ 2008 [7]: 15-0-1 (Document) Exceptions shall only be used for error handling.
- C++ Core Guidelines [11]: E.3: Use exceptions for error handling only
- Effective Java 2nd Edition [15]: Item 57: Use exceptions only for exceptional conditions
- The C++ Programming Language [14], 13.1.1. Exceptions

Rule A15-0-2 (required, architecture / design / implementation, partially automated)
At least the basic guarantee for exception safety shall be provided for all operations. In addition, each function may offer either the strong guarantee or the nothrow guarantee

Rationale

Exceptions introduce additional data flow into a program. It is important to consider all the effects of code taking such paths to always recover from an exception error properly and always preserve object's invariants.

"Well-designed functions are exception safe, meaning they offer at least the basic exception safety guarantee (i.e., the basic guarantee). Such functions assure callers that even if an exception is thrown, program invariants remain intact (i.e., no data structures are corrupted) and no resources are leaked. Functions offering the strong exception safety guarantee (i.e., the strong guarantee) assure callers that if an exception arises, the state of the program remains as it was prior to the call." [effective modern c++]

The C++ standard library always provides one of the following guarantees for its operations, the same needs to be followed by code compliant to the guidelines. "

- Basic guarantee for all operations: The basic invariants of all objects are maintained, and no resources, such as memory, are leaked. In particular, the basic invariants of every built-in and standard-library type guarantee that you can destroy an object or assign to it after every standard-library operation
- Strong guarantee for key operations: in addition to providing the basic guarantee, either the operation succeeds, or it has no effect.
- Nothrow guarantee for some operations: in addition to providing the basic guarantee, some operations are guaranteed not to throw any exception.

” [C++ Programming Reference]

Nothrow means in this context that the function not only does not exit with an exception, but also that internally an exception cannot occur.

Example

```
//% $Id: A15-0-2.cpp 289436 2017-10-04 10:45:23Z michal.szczepankiewicz $
#include <cstdint>
#include <cstring>
class C1
{
  public:
    C1(const C1& rhs)
    {
        CopyBad(rhs);   // Non-compliant - if an exception is thrown, an object
                        // will be left in an indeterminate state
        CopyGood(rhs);  // Compliant - full object will be properly copied or
                        // none of its properties will be changed
    }
    ~C1() { delete[] e; }
    void CopyBad(const C1& rhs)
    {
        if (this != &rhs)
        {
            delete[] e;
            e = nullptr;  // e changed before the block where an exception can
                          // be thrown
            s = rhs.s;  // s changed before the block where an exception can be
                        // thrown

            if (s > 0)
            {
                e = new std::int32_t[s];  // If an exception will be thrown
                                          // here, the
                // object will be left in an indeterminate
                // state
                std::memcpy(e, rhs.e, s * sizeof(std::int32_t));
            }
        }
    }
    void CopyGood(const C1& rhs)
    {
        std::int32_t* eTmp = nullptr;

        if (rhs.s > 0)
        {
            eTmp = new std::int32_t[rhs.s];  // If an exception will be thrown
                                             // here, the
                                             // object will be left unchanged
            std::memcpy(eTmp, rhs.e, rhs.s * sizeof(std::int32_t));
        }
```

```
46
47          delete[] e;
48          e = eTmp;
49          s = rhs.s;
50      }
51
52    private:
53      std::int32_t* e;
54      std::size_t s;
55  };
56  class A
57  {
58    public:
59      A() = default;
60  };
61  class C2
62  {
63    public:
64      C2() : a1(new A), a2(new A)  // Non-compliant - if a2 memory allocation
65                                   // fails, a1 will never be deallocated
66      {
67      }
68
69    private:
70      A* a1;
71      A* a2;
72  };
73  class C3
74  {
75    public:
76      C3() : a1(nullptr), a2(nullptr)  // Compliant
77      {
78          try
79          {
80              a1 = new A;
81              a2 = new A;  // If memory allocation for a2 fails, catch-block will
82                           // deallocate a1
83          }
84
85          catch (...)
86          {
87              delete a1;
88              a1 = nullptr;
89              delete a2;
90              a2 = nullptr;
91              throw;
92          }
93      }
94
95    private:
96      A* a1;
```

```
97      A* a2;
98  };
```

See also

- SEI CERT C++ [10]: ERR56-CPP. Guarantee exception safety

Rule A15-0-3 (required, implementation, non-automated)
Exception safety guarantee of a called function shall be considered.

Rationale

Supplying an external function with an object that throws an exception on specific operations (e.g. in special member functions) may lead to function's unexpected behavior.

Note that the result of a function call supplied with an object which throws on specific operations may differ when the function guarantees the basic exception safety and the strong exception safety.

Example

```
1  //% $Id: A15-0-3.cpp 271687 2017-03-23 08:57:35Z piotr.tanski $
2  #include <cstdint>
3  #include <stdexcept>
4  #include <vector>
5  class A
6  {
7    public:
8      explicit A(std::int32_t value) noexcept(false) : x(value)
9      {
10         if (x == 0)
11         {
12             throw std::invalid_argument("Constructor: Invalid Argument");
13         }
14     }
15
16   private:
17     std::int32_t x;
18 };
19 int main(int, char**)
20 {
21     constexpr std::int32_t limit = 10;
22     std::vector<A> vec1;  // Constructor and assignment operator of A class
23                           // throw exceptions
24
25     try
26     {
27         for (std::int32_t i = 1; i < limit; ++i)
28         {
```

```
29              vec1.push_back(A(i));  // Constructor of A class will not throw for
30                                     // value from 1 to 10
31          }
32
33          vec1.emplace(vec1.begin(),
34                       0);  // Non-compliant - constructor A(0) throws in an
35                            // emplace() method of std::vector. This leads to
36                            // unexpected result of emplace() method. Throwing an
37                            // exception inside an object constructor in emplace()
38                            // leads to duplication of one of vector's elements.
39          // Vector invariants are valid and the object is destructible.
40      }
41      catch (std::invalid_argument& e)
42      {
43          // Handle an exception
44      }
45
46      std::vector<A> vec2;
47      vec2.reserve(limit);
48      try
49      {
50          for (std::int32_t i = limit - 1; i >= 0; --i)
51          {
52              vec2.push_back(A(i));  // Compliant - constructor of A(0) throws for
53                                     // i = 0, but in this case strong exception
54                                     // safety is guaranteed. While push_back()
55                                     // offers strong exception safety guarantee,
56                                     // push_back can only succeed to add a new
57                                     // element or fails and does not change the
58                                     // container
59          }
60      }
61      catch (std::invalid_argument& e)
62      {
63          // Handle an exception
64      }
65
66      return 0;
67  }
```

See also

- none

Rule A15-0-4 (required, architecture / design / implementation, non-automated)
Unchecked exceptions shall be used to represent errors from which the caller cannot reasonably be expected to recover.

Rationale

Problems that are unpreventable and not expected by the caller are represented with instances of unchecked exceptions category. Such problems include:

- Software errors, i.e. preconditions/postconditions violations, arithmetic errors, failed assertions, sanity checks or invalid variable access, that in C++ are typically represented by logic_error, bad_exception, bad_cast and bad_typeid exceptions or their subclasses
- Internal errors of the executable (like VirtualMachineError of Java language), that in C++ are represented by bad_alloc and bad_array_new_length exceptions

It is not possible to recover from such errors in a meaningful way.

Example

```
1  //% $Id: A15-0-4.cpp 289436 2017-10-04 10:45:23Z michal.szczepankiewicz $
2  #include <cstdint>
3  #include <stdexcept>
4  #include <vector>
5  class InvalidArguments : public std::logic_error  // Compliant - invalid
6                                                    // arguments error is
7                                                    // "unchecked" exception
8  {
9    public:
10     using std::logic_error::logic_error;
11 };
12 class OutOfMemory : public std::bad_alloc  // Compliant - insufficient memory
13                                            // error is "unchecked" exception
14 {
15   public:
16     using std::bad_alloc::bad_alloc;
17 };
18 class DivisionByZero : public std::logic_error  // Compliant - division by zero
19                                                 // error is "unchecked"
20                                                 // exception
21 {
22   public:
23     using std::logic_error::logic_error;
24 };
25 class CommunicationError : public std::logic_error  // Non-compliant -
26                                                     // communication error
27                                                     // should be "checked"
28 // exception but defined to be "unchecked"
29 {
30   public:
31     using std::logic_error::logic_error;
32 };
33 double Division(std::int32_t a, std::int32_t b) noexcept(false)
34 {
35     // ...
36     if (b == 0)
```

```
37      {
38          throw DivisionByZero(
39              "Division by zero error");  // Unchecked exception thrown correctly
40      }
41
42      // ...
43  }
44  void Allocate(std::uint32_t bytes) noexcept(false)
45  {
46      // ...
47      throw OutOfMemory();  // Unchecked exception thrown correctly
48  }
49  void InitializeSocket() noexcept(false)
50  {
51      bool validParameters = true;
52
53      // ...
54      if (!validParameters)
55      {
56          throw InvalidArguments("Invalid parameters passed");  // Unchecked
57                                                                // exception
58                                                                // thrown
59                                                                // correctly
60      }
61  }
62  void SendData(std::int32_t socket) noexcept(false)
63  {
64      // ...
65      bool isSentSuccessfully = true;
66
67      // ...
68      if (!isSentSuccessfully)
69      {
70          throw CommunicationError("Could not send data");  // Unchecked exception
71                                                            // thrown when checked
72                                                            // exception should
73                                                            // be.
74      }
75  }
76  void IterateOverContainer(const std::vector<std::int32_t>& container,
77                            std::uint64_t length) noexcept(false)
78  {
79      for (std::uint64_t idx{0U}; idx < length; ++idx)
80      {
81          int32_t value = container.at(idx);  // at() throws std::out_of_range
82                                              // exception when passed integer
83                                              // exceeds the size of container.
84                                              // Unchecked exception thrown
85                                              // correctly
86      }
87  }
```

See also

- Effective Java: Item 58: Use checked exceptions for recoverable conditions and runtime exceptions for programming errors, Item 60: Favor the use of standard exceptions

Rule A15-0-5 (required, architecture / design / implementation, non-automated)
Checked exceptions shall be used to represent errors from which the caller can reasonably be expected to recover.

Rationale

All expected by the caller, but also reasonable to recover from, problems are represented with instances of checked exceptions. Such problems include input/output and other application's runtime errors. It is possible to handle such errors in a meaningful way.

"Overuse of checked exceptions can make an API far less pleasant to use. If a method throws one or more checked exceptions, the code that invokes the method must handle the exceptions in one or more catch blocks, or it must declare that it throws the exceptions and let them propagate outward. Either way, it places a nontrivial burden on the programmer.

The burden is justified if the exceptional condition cannot be prevented by proper use of the API and the programmer using the API can take some useful action once confronted with the exception. Unless both of these conditions hold, an unchecked exception is more appropriate." [Effective Java 2nd Edition [15]]

Example

```
1  //% $Id: A15-0-5.cpp 309502 2018-02-28 09:17:39Z michal.szczepankiewicz $
2  #include <cstdint>
3  #include <stdexcept>
4  #include <system_error>
5
6  // @checkedException
7  class CommunicationError
8      : public std::exception  // Compliant - communication error is "checked"
9  {
10   public:
11     explicit CommunicationError(const char* message) : msg(message) {}
12     CommunicationError(CommunicationError const&) noexcept = default;
13     CommunicationError& operator=(CommunicationError const&) noexcept = default;
14     ~CommunicationError() override = default;
15
16     const char* what() const noexcept override { return msg; }
17
18   private:
19     const char* msg;
```

```
};

// @checkedException
class BusError
    : public CommunicationError  // Compliant - bus error is "checked"
{
  public:
    using CommunicationError::CommunicationError;
};

// @checkedException
class Timeout : public std::runtime_error  // Compliant - communication timeout
                                           // is "checked"
{
  public:
    using std::runtime_error::runtime_error;
};

// @checkedException
class PreconditionsError : public std::exception  // Non-compliant - error on
                                                  // preconditions check should
                                                  // be "unchecked" but is
                                                  // defined to be "checked"
{
    // Implementation
};

void Fn1(std::uint8_t* buffer, std::uint8_t bufferLength) noexcept(false)
{
    bool sentSuccessfully = true;

    // ...
    if (!sentSuccessfully)
    {
        throw CommunicationError(
            "Could not send data");  // Checked exception thrown correctly
    }
}
void Fn2(std::uint8_t* buffer, std::uint8_t bufferLength) noexcept(false)
{
    bool initSuccessfully = true;

    if (!initSuccessfully)
    {
        throw PreconditionsError();  // An exception thrown on preconditions
                                     // check failure should be "Unchecked", but
                                     // PreconditionsError is "Checked"
    }

    // ...
    bool sentSuccessfully = true;
```

```
71      bool isTimeout = false;
72
73      // ...
74      if (!sentSuccessfully)
75      {
76          throw BusError(
77              "Could not send data");  // Checked exception thrown correctly
78      }
79
80      // ...
81      if (isTimeout)
82      {
83          throw Timeout("Timeout reached");  // Checked exception thrown correctly
84      }
85  }
86  void Fn3(std::uint8_t* buffer) noexcept(false)
87  {
88      bool isResourceBusy = false;
89
90      // ...
91      if (isResourceBusy)
92      {
93          throw std::runtime_error(
94              "Resource is busy now");  // Checked exception thrown correctly
95      }
96  }
97  class Thread  // Class which mimics the std::thread
98  {
99    public:
100     // Implementation
101
102     Thread() noexcept(false)
103     {
104         bool resourcesAvailable = false;
105         // ...
106         if (!resourcesAvailable)
107         {
108             throw std::system_error(
109                 static_cast<int>(std::errc::resource_unavailable_try_again),
110                 std::generic_category());  // Compliant - correct usage of
111                                            // checked exception system_error
112         }
113     }
114 };
```

See also

- Effective Java: Item 58 - Use checked exceptions for recoverable conditions and runtime exceptions for programming errors.

Rule A15-0-6 (required, verification / toolchain, non-automated)
An analysis shall be performed to analyze the failure modes of exception handling. In particular, the following failure modes shall be analyzed: (a) worst time execution time not existing or cannot be determined, (b) stack not correctly unwound, (c) exception not thrown, other exception thrown, wrong catch activated, (d) memory not available while exception handling.

Rationale

Note that the worst-case execution time and behavior of exception handling can be hardware specific. This rule requires only that the exception handling is deterministic in the sense that it has a deterministic behavior.

Note: this analysis can be performed by the compiler supplier or it can be done by the project.

See also

- none

Rule A15-0-7 (required, verification / toolchain, partially automated)
Exception handling mechanism shall guarantee a deterministic worst-case time execution time.

Rationale

Compilers, i.e. GCC or Clang, uses dynamic memory allocation in order to allocate currently thrown exception in their exception handling mechanism implementations. This causes a non-deterministic execution time and run-time allocation errors. A possible working approach is to modify the memory allocator so that the dynamic memory does not need to be obtained (from OS) when an exception is thrown.

A static code analysis can search for a use of dynamic memory in the implementation of the try/catch mechanism of the compiler, to show if worst-case time cannot be ensured.

GCC compiler uses following gcc library's functions to provide exception handling mechanism routines:

- __cxa_allocate_exception
- __cxa_throw
- __cxa_free_exception
- __cxa_begin_catch
- __cxa_end_catch
- Specific stack unwinding functions, i.e. _Unwind_RaiseException, _Unwind_Resume, _Unwind_DeleteException, etc.

Example

```
//% $Id: A15-0-7.cpp 289436 2017-10-04 10:45:23Z michal.szczepankiewicz $
#include <cstdlib>
#include <cstring>
struct CxaException
{
    // Exception's structure implementation
};
extern "C" void FatalError(const char* msg)
{
    // Reports an error and terminates the program
}

extern "C" void* CxaAllocateExceptionDynamically(size_t thrownSize)
{
    size_t size = thrownSize + sizeof(CxaException);
    CxaException* buffer = static_cast<CxaException*>(
        malloc(size));  // Non-compliant - dynamic memory allocation used

    if (!buffer)
    {
        FatalError("Not enough memory to allocate exception!");
    }

    memset(buffer, 0, sizeof(CxaException));
    return buffer + 1;
}
extern "C" void* StaticMalloc(size_t size)
{
    void* mem = NULL;
    // Allocates memory using static memory pool
    return mem;
}
extern "C" void* CxaAllocateExceptionStatically(size_t thrownSize)
{
    size_t size = thrownSize + sizeof(CxaException);
    CxaException* buffer = static_cast<CxaException*>(StaticMalloc(
        size));  // Compliant - memory allocation on static memory pool used

    if (!buffer)
    {
        FatalError("Not enough memory to allocate exception!");
    }

    memset(buffer, 0, sizeof(CxaException));
    return buffer + 1;
}
```

See also

- none

Rule A15-0-8 (required, verification / toolchain, non-automated)
A worst-case execution time (WCET) analysis shall be performed to determine maximum execution time constraints of the software, covering in particular the exceptions processing.

Rationale

Some systems require a guarantee that an action will be performed within predictable time constraints. Such real-time systems are allowed to use exception handling mechanism only if there is a tool support for accurate predicting such maximum time boundaries.

“Before deciding that you cannot afford or don’t like exception-based error handling, have a look at the alternatives; they have their own complexities and problems. Also, as far as possible, measure before making claims about efficiency.” [C++ Core Guidelines]

See also

- MISRA C++ 2008 [7]: 15-0-1 (Document) Exceptions shall only be used for error handling.
- open-std.org [18]: ISO/IEC TR 18015:2006(E). Technical Report on C++ Performance

6.15.1 Throwing an exception

Rule A15-1-1 (advisory, implementation, automated)
Only instances of types derived from std::exception should be thrown.

Rationale

If an object that inherits from std::exception is thrown, there’s a guarantee that it serves to document the cause of an exception in an unified way. Also, "it makes your code easier to learn and re-use, because it matches established conventions with which programmers are already familiar.". [Effective Java 2nd Edition [15]]

This means that only standard library exceptions or user-defined exceptions that inherit from std::exception base class should be used for exceptions.

Note that direct instances of std::exception are not to be thrown as they can not be unique.

Example

```
1 //% $Id: A15-1-1.cpp 289436 2017-10-04 10:45:23Z michal.szczepankiewicz $
2 #include <memory>
3 #include <stdexcept>
4 class A
```

```
{
    // Implementation
};
class MyException : public std::logic_error
{
  public:
    using std::logic_error::logic_error;
    // Implementation
};
void F1()
{
    throw - 1;  // Non-compliant - integer literal thrown
}
void F2()
{
    throw nullptr;  // Non-compliant - null-pointer-constant thrown
}
void F3()
{
    throw A();  // Non-compliant - user-defined type that does not inherit from
                // std::exception thrown
}
void F4()
{
    throw std::logic_error{
        "Logic Error"};  // Compliant - std library exception thrown
}
void F5()
{
    throw MyException{"Logic Error"};  // Compliant - user-defined type that
                                       // inherits from std::exception thrown
}
void F6()
{
    throw std::make_shared<std::exception>(
        std::logic_error("Logic Error"));  // Non-compliant - shared_ptr does
                                           // not inherit from std::exception
}
void F7()
{
    try
    {
        F6();
    }

    catch (std::exception& e)  // An exception of
                               // std::shared_ptr<std::exception> type will not
                               // be caught here
    {
        // Handle an exception
    }
```

```
56      catch (std::shared_ptr<std::exception>& e)  // An exception of
57      // std::shared_ptr<std::exception>
58      // type will be caught here, but
59      // unable to access
60      // std::logic_error information
61      {
62          // Handle an exception
63      }
64  }
```

See also

- HIC++ v4.0 [9]: 15.1.1 Only use instances of std::exception for exceptions
- C++ Core Guidelines [11]: E.14: Use purpose-designed user-defined types as exceptions (not built-in types)
- Effective Java 2nd Edition [15]: Item 60: Favor the use of standard exceptions

Rule A15-1-2 (required, implementation, automated)
An exception object shall not be a pointer.

Rationale

If an exception object of pointer type is thrown and that pointer refers to a dynamically created object, then it may be unclear which function is responsible for destroying it, and when. This may lead to memory leak.

If an exception object of pointer type is thrown and that pointer refers to an automatic variable, it allows using a variable after its destruction, leading to undefined behavior.

This ambiguity does not exist if a copy of the object is thrown.

Example

```
1   //% $Id: A15-1-2.cpp 289436 2017-10-04 10:45:23Z michal.szczepankiewicz $
2   #include <cstdint>
3   class A
4   {
5       // Implementation
6   };
7   void Fn(std::int16_t i)
8   {
9       A a1;
10      A& a2 = a1;
11      A* a3 = new A;
12
13      if (i < 10)
14      {
15          throw a1;  // Compliant - copyable object thrown
16      }
```

```
17
18      else if (i < 20)
19      {
20          throw A();  // Compliant - copyable object thrown
21      }
22
23      else if (i < 30)
24      {
25          throw a2;  // Compliant - copyable object thrown
26      }
27
28      else if (i < 40)
29      {
30          throw & a1;  // Non-compliant - pointer type thrown
31      }
32
33      else if (i < 50)
34      {
35          throw a3;  // Non-compliant - pointer type thrown
36      }
37
38      else if (i < 60)
39      {
40          throw(*a3);  // Compliant - memory leak occurs, violates other rules
41      }
42
43      else
44      {
45          throw new A;  // Non-compliant - pointer type thrown
46      }
47  }
```

See also

- MISRA C++ 2008 [7]: 15-0-2 An exception object should not have pointer type.
- C++ Core Guidelines [11]: E.13: Never throw while being the direct owner of an object

Rule M15-0-3 (required, implementation, automated)
Control shall not be transferred into a try or catch block using a goto or a switch statement.

See MISRA C++ 2008 [7]

Rule M15-1-1 (required, implementation, automated)
The assignment-expression of a throw statement shall not itself cause an exception to be thrown.

See MISRA C++ 2008 [7]

Rule M15-1-2 (required, implementation, automated)
NULL shall not be thrown explicitly.

See MISRA C++ 2008 [7]

Rule M15-1-3 (required, implementation, automated)
An empty throw (throw;) shall only be used in the compound statement of a catch handler.

See MISRA C++ 2008 [7]

Rule A15-1-3 (advisory, implementation, automated)
All thrown exceptions should be unique.

Rationale

Defining unique exceptions in the project significantly simplifies debug process.

An exception is considered to be unique if at least one of the following conditions is fulfilled:

- The type of the exception does not occur in any other place in the project
- The error message (i.e. message itself, error code, etc.) of the exception does not occur in any other place in the project

Example

```
1  //% $Id: A15-1-3.cpp 289436 2017-10-04 10:45:23Z michal.szczepankiewicz $
2  #include <iostream>
3  #include <sstream>
4  #include <stdexcept>
5  #include <string>
6  static std::string ComposeMessage(const char* file,
7                                    const char* func,
8                                    std::int32_t line,
9                                    const std::string& message) noexcept
10 {
11     std::stringstream s;
12     s << "(" << file << ", " << func << ":" << line << "): " << message;
```

```
13      return s.str();
14  }
15  void F1()
16  {
17      // ...
18      throw std::logic_error("Error");
19  }
20  void F2()
21  {
22      // ...
23      throw std::logic_error("Error");  // Non-compliant - both exception type and
24                                        // error message are not unique
25  }
26  void F3()
27  {
28      // ...
29      throw std::invalid_argument(
30          "Error");  // Compliant - exception type is unique
31  }
32  void F4() noexcept(false)
33  {
34      // ...
35      throw std::logic_error("f3(): preconditions were not met");  // Compliant -
36                                                                   // error
37                                                                   // message is
38                                                                   // unique
39  }
40  void F5() noexcept(false)
41  {
42      // ...
43      throw std::logic_error(ComposeMessage(
44          __FILE__,
45          __func__,
46          __LINE__,
47          "postconditions were not met"));  // Compliant - error message is unique
48  }
49  void F6() noexcept
50  {
51      try
52      {
53          F1();
54          F2();
55          F3();
56      }
57
58      catch (std::invalid_argument& e)
59      {
60          std::cout << e.what() << '\n';  // Only f3() throws this type of
61                                          // exception, it is easy to deduce which
62                                          // function threw
63      }
```

```
64
65      catch (std::logic_error& e)
66      {
67          std::cout << e.what() << '\n';  // f1() and f2() throw exactly the same
68                                          // exceptions, unable to deduce which
69                                          // function threw
70      }
71
72      try
73      {
74          F4();
75          F5();
76      }
77
78      catch (std::logic_error& e)
79      {
80          std::cout << e.what() << '\n';  // Debugging process simplified, because
81                                          // of unique error message it is known
82                                          // which function threw
83      }
84  }
```

See also

- Effective Java 2nd Edition [15]: Item 63: Include failure-capture information in detail messages

Rule A15-1-4 (required, implementation, partially automated)
If a function exits with an exception, then before a throw, the function shall place all objects/resources that the function constructed in valid states or it shall delete them.

Rationale

If the only handler to dynamically allocated memory or system resource (e.g. file, lock, network connection or thread) goes out of scope due to throwing an exception, memory leak occurs. Memory leaks lead to performance degradation, security violations and software crashes.

Allocated memory or system resource can be released by explicit call to resource deinitialization or memory deallocation function (such as operator delete), before each return/try/break/continue statement. However, this solution is error prone and difficult to maintain.

The recommended way of releasing dynamically allocated objects and resources is to follow RAII ("'Resource Acquisition Is Initialization"') design pattern, also known as Scope-Bound Resource Management or "Constructor Acquires, Destructor Releases" (CADRe). It allows to bind the life cycle of the resource to the lifetime of a scope-bound object. It guarantees that resources are properly deinitialized and released when data flow reaches the end of the scope.

Examples of RAII design pattern that significantly simplifies releasing objects/resources on throwing an exception are C++ smart pointers: std::unique_ptr and std::shared_ptr.

Example

```
//% $Id: A15-1-4.cpp 289436 2017-10-04 10:45:23Z michal.szczepankiewicz $
#include <cstdint>
#include <memory>
#include <stdexcept>
extern std::uint32_t F1();
void FVeryBad() noexcept(false)
{
    std::logic_error* e = new std::logic_error("Logic Error 1");
    // ...
    std::uint32_t i = F1();

    if (i < 10)
    {
        throw(*e);  // Non-compliant - fVeryBad() is not able to clean-up
                    // allocated memory
    }

    // ...
    delete e;
}
void FBad() noexcept(false)
{
    std::int32_t* x = new std::int32_t(0);
    // ...
    std::uint32_t i = F1();

    if (i < 10)
    {
        throw std::logic_error("Logic Error 2");  // Non-compliant - exits from
                                                  // fBad() without cleaning-up
                                                  // allocated resources and
                                                  // causes a memory leak
    }

    else if (i < 20)
    {
        throw std::runtime_error("Runtime Error 3");  // Non-compliant - exits
                                                      // from fBad() without
                                                      // cleaning-up allocated
                                                      // resources and causes a
                                                      // memory leak
    }

    // ...
    delete x;  // Deallocates claimed resource only in the end of fBad() scope
}
```

```
47 void FGood() noexcept(false)
48 {
49     std::int32_t* y = new std::int32_t(0);
50     // ...
51     std::uint32_t i = F1();
52
53     if (i < 10)
54     {
55         delete y;  // Deletes allocated resource before throwing an exception
56         throw std::logic_error("Logic Error 4");  // Compliant - deleting y
57                                                   // variable before exception
58                                                   // leaves the fGood() scope
59     }
60
61     else if (i < 20)
62     {
63         delete y;  // Deletes allocated resource before throwing an exception
64         throw std::runtime_error("Runtime Error 5");  // Compliant - deleting y
65                                                       // variable before
66                                                       // exception leaves the
67                                                       // fGood() scope
68     }
69
70     else if (i < 30)
71     {
72         delete y;  // Deletes allocated resource before throwing an exception
73                    // again, difficult to maintain
74         throw std::invalid_argument(
75             "Invalid Argument 1");  // Compliant - deleting
76                                     // y variable before
77                                     // exception leaves the
78                                     // fGood() scope
79     }
80
81     // ...
82     delete y;  // Deallocates claimed resource also in the end of fGood() scope
83 }
84 void FBest() noexcept(false)
85 {
86     std::unique_ptr<std::int32_t> z = std::make_unique<std::int32_t>(0);
87     // ...
88     std::uint32_t i = F1();
89
90     if (i < 10)
91     {
92         throw std::logic_error("Logic Error 6");  // Compliant - leaving the
93                                                   // fBest() scope causes
94                                                   // deallocation of all
95         // automatic variables, unique_ptrs, too
96     }
97
```

```
98      else if (i < 20)
99      {
100         throw std::runtime_error("Runtime Error 3");  // Compliant - leaving the
101                                                       // fBest() scope causes
102                                                       // deallocation of all
103                                                       // automatic variables,
104                                                       // unique_ptrs, too
105     }
106
107     else if (i < 30)
108     {
109         throw std::invalid_argument(
110             "Invalid Argument 2");  // Compliant - leaving the fBest() scope
111                                     // causes deallocation of all automatic
112                                     // variables, unique_ptrs,
113         // too
114     }
115
116     // ...
117     // z is deallocated automatically here, too
118 }
119 class CRaii  // Simple class that follows RAII pattern
120 {
121   public:
122     CRaii(std::int32_t* pointer) noexcept : x(pointer) {}
123     ~CRaii()
124     {
125         delete x;
126         x = nullptr;
127     }
128
129   private:
130     std::int32_t* x;
131 };
132 void FBest2() noexcept(false)
133 {
134     CRaii a1(new std::int32_t(10));
135     // ...
136     std::uint32_t i = F1();
137
138     if (i < 10)
139     {
140         throw std::logic_error("Logic Error 7");  // Compliant - leaving the
141                                                   // fBest2() scope causes a1
142                                                   // variable deallocation
143                                                   // automatically
144     }
145     else if (i < 20)
146     {
147         throw std::runtime_error("Runtime Error 4");  // Compliant - leaving the
148                                                       // fBest2() scope causes
```

```
149                                                     // a1 variable
150                                                     // deallocation
151                                                     // automatically
152     }
153     else if (i < 30)
154     {
155         throw std::invalid_argument(
156             "Invalid Argument 3");  // Compliant - leaving the fBest2() scope
157                                     // causes a1 variable deallocation
158                                     // automatically
159     }
160
161     // ...
162     // a1 is deallocated automatically here, too
163 }
```

See also

- SEI CERT C++ [10]: ERR57-CPP. Do not leak resources when handling exceptions
- C++ Core Guidelines [11]: E.6: Use RAII to prevent leaks.

Rule A15-1-5 (required, implementation, non-automated)
Exceptions shall not be thrown across execution boundaries.

Rationale

An execution boundary is the delimitation between code compiled by differing compilers, including different versions of a compiler produced by the same vendor. For instance, a function may be declared in a header file but defined in a library that is loaded at runtime. The execution boundary is between the call site in the executable and the function implementation in the library. Such boundaries are also called ABI (application binary interface) boundaries because they relate to the interoperability of application binaries.

Throwing an exception across an execution boundary requires that both sides of the execution boundary use the same ABI for exception handling, which may be difficult to ensure.

Exception

If it can be ensured that the execution boundaries use the same ABI for exception handling routines on both sides, then throwing an exception across these execution boundaries is allowed.

See also

- SEI CERT C++ [10]: ERR59-CPP. Do not throw an exception across execution boundaries

6.15.2 Constructors and destructors

Rule A15-2-1 (required, implementation, automated)
Constructors that are not noexcept shall not be invoked before program startup.

Rationale

Before the program starts executing the body of main function, it is in a start-up phase, constructing and initializing static objects. There is nowhere an exception handler can be placed to catch exceptions thrown during this phase, so if an exception is thrown it leads to the program being terminated in an implementation-defined manner.

Such errors may be more difficult to find because an error message can not be logged, due to lack of exception handling mechanism during static initialization.

Example

```
1  //% $Id: A15-2-1.cpp 271927 2017-03-24 12:01:35Z piotr.tanski $
2  #include <cstdint>
3  #include <stdexcept>
4  class A
5  {
6    public:
7      A() noexcept : x(0)
8      {
9          // ...
10     }
11     explicit A(std::int32_t n) : x(n)
12     {
13         // ...
14         throw std::runtime_error("Unexpected error");
15     }
16     A(std::int32_t i, std::int32_t j) noexcept : x(i + j)
17     {
18         try
19         {
20             // ...
21             throw std::runtime_error("Error");
22             // ...
23         }
24
25         catch (std::exception& e)
26         {
27         }
28     }
29
30   private:
31     std::int32_t x;
32 };
```

```
static A a1;      // Compliant - default constructor of type A is noexcept
static A a2(5);   // Non-compliant - constructor of type A throws, and the
                  // exception will not be caught by the handler in main function
static A a3(5, 10);  // Compliant - constructor of type A is noexcept, it
                     // handles exceptions internally
int main(int, char**)
{
    try
    {
        // program code
    }
    catch (...)
    {
        // Handle exceptions
    }

    return 0;
}
```

See also

- SEI CERT C++ [10]: ERR51-CPP. Handle all exceptions.

Rule A15-2-2 (required, implementation, partially automated)
If a constructor is not noexcept and the constructor cannot finish object initialization, then it shall deallocate the object's resources and it shall throw an exception.

Rationale

Leaving the constructor with invalid object state leads to undefined behavior.

Example

```
//% $Id: A15-2-2.cpp 289436 2017-10-04 10:45:23Z michal.szczepankiewicz $
#include <fstream>
#include <stdexcept>
class A
{
  public:
    A() = default;
};
class C1
{
  public:
    C1()
    noexcept(false)
            : a1(new A), a2(new A)   // Non-compliant - if a2 memory allocation
                                     // fails, a1 will never be deallocated
    {
```

```
17      }
18      C1(A* pA1, A* pA2)
19      noexcept : a1(pA1), a2(pA2)  // Compliant - memory allocated outside of C1
20                                   // constructor, and no exceptions can be thrown
21      {
22      }
23
24    private:
25      A* a1;
26      A* a2;
27  };
28  class C2
29  {
30    public:
31      C2() noexcept(false) : a1(nullptr), a2(nullptr)
32      {
33          try
34          {
35              a1 = new A;
36              a2 = new A;  // If memory allocation for a2 fails, catch-block will
37              // deallocate a1
38          }
39
40          catch (std::exception& e)
41          {
42              throw;  // Non-compliant - whenever a2 allocation throws an
43                      // exception, a1 will never be deallocated
44          }
45      }
46
47    private:
48      A* a1;
49      A* a2;
50  };
51  class C3
52  {
53    public:
54      C3() noexcept(false) : a1(nullptr), a2(nullptr), file("./filename.txt")
55      {
56          try
57          {
58              a1 = new A;
59              a2 = new A;
60
61              if (!file.good())
62              {
63                  throw std::runtime_error("Could not open file.");
64              }
65          }
66
67          catch (std::exception& e)
```

```
68          {
69              delete a1;
70              a1 = nullptr;
71              delete a2;
72              a2 = nullptr;
73              file.close();
74              throw;  // Compliant - all resources are deallocated before the
75                      // constructor exits with an exception
76          }
77      }
78
79    private:
80      A* a1;
81      A* a2;
82      std::ofstream file;
83  };
84  class C4
85  {
86    public:
87      C4() : x(0), y(0)
88      {
89          // Does not need to check preconditions here - x and y initialized with
90          // correct values
91      }
92      C4(std::int32_t first, std::int32_t second)
93      noexcept(false) : x(first), y(second)
94      {
95          CheckPreconditions(x,
96                             y);  // Compliant - if constructor failed to create a
97                                  // valid object, then throw an exception
98      }
99      static void CheckPreconditions(std::int32_t x,
100                                     std::int32_t y) noexcept(false)
101     {
102         if (x < 0 || x > 1000)
103         {
104             throw std::invalid_argument(
105                 "Preconditions of class C4 were not met");
106         }
107
108         else if (y < 0 || y > 1000)
109         {
110             throw std::invalid_argument(
111                 "Preconditions of class C4 were not met");
112         }
113     }
114
115   private:
116     std::int32_t x;  // Acceptable range: <0; 1000>
117     std::int32_t y;  // Acceptable range: <0; 1000>
118 };
```

See also

- C++ Core Guidelines [11]: C.42: If a constructor cannot construct a valid object, throw an exception

6.15.3 Handling an exception

Rule M15-3-1 (required, implementation, automated)
Exceptions shall be raised only after start-up and before termination.

See MISRA C++ 2008 [7]

Rule A15-3-2 (required, implementation, non-automated)
If a function throws an exception, it shall be handled when meaningful actions can be taken, otherwise it shall be propagated.

Rationale

Provide exception handlers only for functions that actually are able to take recovery or cleanup actions. Implementing meaningless exception handlers that only re-throw caught exception results in an implementation that is inefficient and difficult to maintain.

Example

```
1  //% $Id: A15-3-2.cpp 309502 2018-02-28 09:17:39Z michal.szczepankiewicz $
2  #include <cstdint>
3  #include <iostream>
4  #include <stdexcept>
5  #include <memory>
6
7  /// @checkedException
8  class CommunicationError : public std::exception
9  {
10     // Implementation
11 };
12
13 /// @throw CommunicationError Exceptional communication errors
14 extern void Send(std::uint8_t* buffer) noexcept(false);
15
16 void SendData1(std::uint8_t* data) noexcept(false)
17 {
18     try
19     {
20         Send(data);
21     }
22
23     catch (CommunicationError& e)
```

```
24      {
25          std::cerr << "Communication error occured" << std::endl;
26          throw;  // Non-compliant - exception is not handled, just re-thrown
27      }
28  }
29  extern void BusRestart() noexcept;
30  extern void BufferClean() noexcept;
31  void SendData2(std::uint8_t* data) noexcept(false)
32  {
33      try
34      {
35          Send(data);
36      }
37
38      catch (CommunicationError& e)
39      {
40          std::cerr << "Communication error occured" << std::endl;
41          BufferClean();
42          throw;  // Compliant - exception is partially handled and re-thrown
43      }
44  }
45  void F1() noexcept
46  {
47      std::uint8_t* buffer = nullptr;
48
49      // ...
50      try
51      {
52          SendData2(buffer);
53      }
54
55      catch (CommunicationError& e)
56      {
57          std::cerr << "Communication error occured" << std::endl;
58          BusRestart();
59          // Compliant - including SendData2() exception handler, exception is now
60          // fully handled
61      }
62  }
63  void SendData3(std::uint8_t* data) noexcept
64  {
65      try
66      {
67          Send(data);
68      }
69
70      catch (CommunicationError& e)
71      {
72          std::cerr << "Communication error occured" << std::endl;
73          BufferClean();
74          BusRestart();
```

```
75          // Compliant - exception is fully handled
76      }
77  }
78
79  struct A
80  {
81      std::uint32_t x;
82  };
83
84  std::unique_ptr<A[]> Func1()
85  {
86      //rather throws std::bad_alloc
87      return std::make_unique<A[]>(9999999999999999);
88  }
89
90  std::unique_ptr<A[]> Func2()
91  {
92      //does not catch std::bad_alloc
93      //because nothing meaningful can be done here
94      return Func1();
95  }
96
97  std::unique_ptr<A[]> Func3()
98  {
99      //does not catch std::bad_alloc
100     //because nothing meaningful can be done here
101     return Func2();
102 }
103
104 extern void Cleanup() noexcept;
105
106 int main(void)
107 {
108     try
109     {
110          Func3();
111     }
112     catch (const std::exception& ex)
113     {
114         //catches std::bad_alloc here and
115         //terminates the application
116         //gracefully
117         Cleanup();
118     }
119
120     return 0;
121 }
```

See also

- none

Rule A15-3-3 (required, implementation, partially-automated)
Main function and a task main function shall catch at least: base class exceptions from all third-party libraries used, std::exception and all otherwise unhandled exceptions.

Rationale

If a program throws an unhandled exception in main function, as well as in init thread function, the program terminates in an implementation-defined manner. In particular, it is implementation-defined whether the call stack is unwound, before termination, so the destructors of any automatic objects may or may not be executed. By enforcing the provision of a "last-ditch catch-all", the developer can ensure that the program terminates in a consistent manner.

Exceptions hierarchy from external libraries may be completely separate from C++ Standard Library `std::exception`. Handling such base exception classes separately may provide some additional information about application termination causes.

Example

```
1  //% $Id: A15-3-3.cpp 309502 2018-02-28 09:17:39Z michal.szczepankiewicz $
2  #include <stdexcept>
3
4  //base exception class from external library that is used
5  class ExtLibraryBaseException {};
6
7  int MainGood(int, char**)  // Compliant
8  {
9      try
10     {
11         // program code
12     }
13     catch (std::runtime_error& e)
14     {
15         // Handle runtime errors
16     }
17     catch (std::logic_error& e)
18     {
19         // Handle logic errors
20     }
21     catch (ExtLibraryBaseException &e)
22     {
23         // Handle all expected exceptions
24         // from an external library
25     }
26     catch (std::exception& e)
27     {
28         // Handle all expected exceptions
29     }
30     catch (...)
31     {
```

```
32          // Handle all unexpected exceptions
33      }
34
35      return 0;
36  }
37  int MainBad(int, char**)  // Non-compliant - neither unexpected exceptions
38                            // nor external libraries exceptions are caught
39  {
40      try
41      {
42          // program code
43      }
44      catch (std::runtime_error& e)
45      {
46          // Handle runtime errors
47      }
48      catch (std::logic_error& e)
49      {
50          // Handle logic errors
51      }
52      catch (std::exception& e)
53      {
54          // Handle all expected exceptions
55      }
56
57      return 0;
58  }
59  void ThreadMainGood()  // Compliant
60  {
61      try
62      {
63          // thread code
64      }
65      catch (ExtLibraryBaseException &e)
66      {
67          // Handle all expected exceptions
68          // from an external library
69      }
70      catch (std::exception& e)
71      {
72          // Handle all expected exception
73      }
74      catch (...)
75      {
76          // Handle all unexpected exception
77      }
78  }
79
80  void ThreadMainBad()      // Non-compliant - neither unexpected exceptions
81                            // nor external libraries exceptions are caught
82  {
```

```
83      try
84      {
85          // thread code
86      }
87      catch (std::exception& e)
88      {
89          // Handle all expected exceptions
90      }
91
92      // Uncaught unexpected exception will cause an immediate program termination
93  }
```

See also

- MISRA C++ 2008 [7]: 15-3-2 There should be at least one exception handler to catch all otherwise unhandled exceptions.
- SEI CERT C++ [10]: ERR51-CPP. Handle all exceptions
- Effective Java 2nd Edition [15]: Item 65: Don't ignore exceptions

Rule A15-3-4 (required, implementation, non-automated)
Catch-all (ellipsis and std::exception) handlers shall be used only in (a) main, (b) task main functions, (c) in functions that are supposed to isolate independent components and (d) when calling third-party code that uses exceptions not according to AUTOSAR C++14 guidelines.

Rationale

Catching an exception through catch-all handlers does not provide any detailed information about caught exception. This does not allow to take meaningful actions to recover from an exception other than to re-throw it. This is inefficient and results in code that is difficult to maintain.

Example

```
1   //% $Id: A15-3-4.cpp 289436 2017-10-04 10:45:23Z michal.szczepankiewicz $
2   #include <stdexcept>
3   #include <thread>
4   extern std::int32_t Fn();  // Prototype of external third-party library function
5   void F1() noexcept(false)
6   {
7       try
8       {
9           std::int32_t ret = Fn();
10          // ...
11      }
12
13      // ...
14      catch (...)  // Compliant
```

```
15      {
16          // Handle all unexpected exceptions from fn() function
17      }
18  }
19  void F2() noexcept(false)
20  {
21      std::int32_t ret =
22          Fn();  // Non-compliant - can not be sure whether fn() throws or not
23
24      if (ret < 10)
25      {
26          throw std::underflow_error("Error");
27      }
28
29      else if (ret < 20)
30      {
31          // ...
32      }
33      else if (ret < 30)
34      {
35          throw std::overflow_error("Error");
36      }
37
38      else
39      {
40          throw std::range_error("Error");
41      }
42  }
43  void F3() noexcept(false)
44  {
45      try
46      {
47          F2();
48      }
49
50      catch (std::exception& e)  // Non-compliant - caught exception is too
51                                 // general, no information which error occured
52      {
53          // Nothing to do
54          throw;
55      }
56  }
57  void F4() noexcept(false)
58  {
59      try
60      {
61          F3();
62      }
63
64      catch (...)  // Non-compliant - no information about the exception
65      {
```

```
66          // Nothing to do
67          throw;
68      }
69  }
70  class ExecutionManager
71  {
72    public:
73      ExecutionManager() = default;
74      void Execute() noexcept(false)
75      {
76          try
77          {
78              F3();
79          }
80
81          // ...
82          catch (std::exception& e)  // Compliant
83          {
84              // Handle all expected exceptions
85          }
86          catch (...)  // Compliant
87          {
88              // Handle all unexpected exceptions
89          }
90      }
91  };
92  void ThreadMain() noexcept
93  {
94      try
95      {
96          F3();
97      }
98
99      // ...
100     catch (std::exception& e)  // Compliant
101     {
102         // Handle all expected exceptions
103     }
104     catch (...)  // Compliant
105     {
106         // Handle all unexpected exceptions
107     }
108 }
109 int main(int, char**)
110 {
111     try
112     {
113         ExecutionManager execManager;
114         execManager.Execute();
115         // ...
116         std::thread t(&ThreadMain);
```

```
117             // ...
118             F4();
119         }
120
121         // ...
122         catch (std::exception& e)  // Compliant
123         {
124             // Handle all expected exceptions
125         }
126         catch (...)  // Compliant
127         {
128             // Handle all unexpected exceptions
129         }
130
131         return 0;
132     }
```

See also

- none

Rule M15-3-3 (required, implementation, automated)
Handlers of a function-try-block implementation of a class constructor or destructor shall not reference non-static members from this class or its bases.

See MISRA C++ 2008 [7]

Rule M15-3-4 (required, implementation, automated)
Each exception explicitly thrown in the code shall have a handler of a compatible type in all call paths that could lead to that point.

See MISRA C++ 2008 [7]

Rule A15-3-5 (required, implementation, automated)
A class type exception shall be caught by reference or const reference.

Rationale

If a class type exception object is caught by value, slicing occurs. That is, if the exception object is of a derived class and is caught as the base, only the base class's functions (including virtual functions) can be called. Also, any additional member data in the derived class cannot be accessed. If the exception is caught by reference or const reference, slicing does not occur.

Example

```
//% $Id: A15-3-5.cpp 289436 2017-10-04 10:45:23Z michal.szczepankiewicz $
#include <iostream>
#include <stdexcept>
class Exception : public std::runtime_error
{
  public:
    using std::runtime_error::runtime_error;
    const char* what() const noexcept(true) override
    {
        return "Exception error message";
    }
};
void Fn()
{
    try
    {
        // ...
        throw std::runtime_error("Error");
        // ...
        throw Exception("Error");
    }

    catch (const std::logic_error& e)  // Compliant - caught by const reference
    {
        // Handle exception
    }
    catch (std::runtime_error& e)  // Compliant - caught by reference
    {
        std::cout << e.what() << "\n";  // "Error" or "Exception error message"
        // will be printed, depending upon the
        // actual type of thrown object
        throw e;  // The exception re-thrown is of its original type
    }

    catch (
        std::runtime_error
            e)  // Non-compliant - derived types will be caught as the base type
    {
        std::cout
            << e.what()
            << "\n";  // Will always call what() method from std::runtime_error
        throw e;  // The exception re-thrown is of the std::runtime_error type,
                  // not the original exception type
    }
}
```

See also

- MISRA C++ 2008 [7]: 15-3-5 A class type exception shall always be caught by reference.
- SEI CERT C++ [10]: ERR61-CPP. Catch exceptions by lvalue reference

- C++ Core Guidelines [11]: E.15: Catch exceptions from a hierarchy by reference

Rule M15-3-6 (required, implementation, automated)
Where multiple handlers are provided in a single try-catch statement or function-try-block for a derived class and some or all of its bases, the handlers shall be ordered most-derived to base class.

See MISRA C++ 2008 [7]

Rule M15-3-7 (required, implementation, automated)
Where multiple handlers are provided in a single try-catch statement or function-try-block, any ellipsis (catch-all) handler shall occur last.

See MISRA C++ 2008 [7]

6.15.4 Exception specifications

Rule A15-4-1 (required, implementation, automated)
Dynamic exception-specification shall not be used.

Rationale

This feature was deprecated in the 2011 C++ Language Standard (See: Deprecating Exception Specifications).

Main issues with dynamic exception specifications are:

1. Run-time checking: Exception specifications are checked at runtime, so the program does not guarantee that all exceptions will be handled. The run-time failure mode does not lend itself to recovery.
2. Run-time overhead: Run-time checking requires the compiler to produce additional code that hampers optimizations.
3. Unusable in generic code: It is not possible to know what types of exceptions may be thrown from templates arguments operations, so a precise exception specification cannot be written.

In place of dynamic exception-specification, use noexcept specification that allows to declare whether a function throws or does not throw exceptions.

Note: std::unexpected_handler shall not be used.

Example

```
//% $Id: A15-4-1.cpp 289436 2017-10-04 10:45:23Z michal.szczepankiewicz $
#include <stdexcept>
void F1() noexcept;  // Compliant - note that noexcept is the same as
                     // noexcept(true)
void F2() throw();   // Non-compliant - dynamic exception-specification is
                     // deprecated
void F3() noexcept(false);            // Compliant
void F4() throw(std::runtime_error);  // Non-compliant - dynamic
// exception-specification is deprecated
void F5() throw(
    ...);  // Non-compliant - dynamic exception-specification is deprecated
template <class T>
void F6() noexcept(noexcept(T()));  // Compliant
```

See also

- C++ Core Guidelines [11]: E.12: Use noexcept when exiting a function because of a throw is impossible or unacceptable
- open-std.org [18]: open std Deprecating Exception Specifications
- mill22: A Pragmatic Look at Exception Specifications

Rule A15-4-2 (required, implementation, automated)
If a function is declared to be noexcept, noexcept(true) or noexcept(<true condition>), then it shall not exit with an exception.

Rationale

If a function declared to be noexcept, noexcept(true) or noexcept(true condition) throws an exception, then std::terminate() is called immediately. It is implementation-defined whether the call stack is unwound before std::terminate() is called.

To ensure that the rule is not violated, if function's noexcept specification can not be determined, then always declare it to be noexcept(false).

Example

```
//% $Id: A15-4-2.cpp 289436 2017-10-04 10:45:23Z michal.szczepankiewicz $
#include <stdexcept>
// library.h
void LibraryFunc();
// project.cpp
void F1() noexcept
{
    // ...
    throw std::runtime_error("Error");  // Non-compliant - f1 declared to be
                                        // noexcept, but exits with exception.
                                        // This leads to std::terminate() call
}
void F2() noexcept(true)
```

```
{
    try
    {
        // ...
        throw std::runtime_error(
            "Error");  // Compliant - exception will not leave f2
    }
    catch (std::runtime_error& e)
    {
        // Handle runtime error
    }
}
void F3() noexcept(false)
{
    // ...
    throw std::runtime_error("Error");  // Compliant
}
void F4() noexcept(
    false)  // Compliant - no information whether library_func() throws or not
{
    LibraryFunc();
}
```

See also

- MISRA C++ 2008 [7]: 15-5-2: Where a function's declaration includes an exception-specification, the function shall only be capable of throwing exceptions of the indicated type(s).
- MISRA C++ 2008 [7]: 15-5-3: The terminate() function shall not be called implicitly.
- HIC++ v4.0 [9]: 15.3.2: Ensure that a program does not result in a call to std::terminate
- SEI CERT C++ Coding Standard [10]: ERR50-CPP: Do not abruptly terminate the program.

Rule A15-4-3 (required, implementation, automated)
The noexcept specification of a function shall either be identical across all translation units, or identical or more restrictive between a virtual member function and an overrider.

Rationale

Declarations of the same function, even in different translation units, have to specify the same noexcept specification. Overriding functions have to specify the same or a stricter noexcept specification than the base class function which they override.

Note that in many cases, a violation of this rule will lead to a compilation error. This is not guaranteed, however, in particular when function declarations appear in separate translation units.

Example

```
//% $Id: A15-4-3.cpp 317753 2018-04-27 07:44:02Z jan.babst $
// f1.hpp
void Fn() noexcept;

// f1.cpp
// #include <f1.hpp>
void Fn() noexcept  // Compliant
{
    // Implementation
}

// f2.cpp
// #include <f1.hpp>
void Fn() noexcept(false)  // Non-compliant - different exception specifier
{
    // Implementation
}

class A
{
  public:
    void F() noexcept;
    void G() noexcept(false);
    virtual void V1() noexcept = 0;
    virtual void V2() noexcept(false) = 0;
};
void A::F() noexcept  // Compliant
// void A::F() noexcept(false)  // Non-compliant - different exception specifier
// than in declaration
{
    // Implementation
}
void A::G() noexcept(false)  // Compliant
// void A::G() noexcept  // Non-compliant - different exception specifier than
// in declaration
{
    // Implementation
}
class B : public A
{
  public:
    void V1() noexcept override  // Compliant
    // void V1() noexcept(false) override  // Non-compliant - less restrictive
    // exception specifier in derived method, non-compilable
    {
        // Implementation
```

```
47      }
48      void V2() noexcept override  // Compliant - stricter noexcept specification
49      {
50          // Implementation
51      }
52  };
```

See also

- MISRA C++ 2008 [7]: 15-4-1: If a function is declared with an exception-specification, then all declarations of the same function (in other translation units) shall be declared with the same set of type-ids.

Rule A15-4-4 (required, implementation, automated)
A declaration of non-throwing function shall contain noexcept specification.

Rationale

Noexcept specification is a method for a programmer to inform the compiler whether or not a function should throw exceptions. The compiler can use this information to enable certain optimizations on non-throwing functions as well as enable the noexcept operator, which can check at compile time if a particular expression is declared to throw any exceptions.

Noexcept specification is also a method to inform other programmers that a function does not throw any exceptions.

A non-throwing function needs to declare noexcept specifier. A function that may or may not throw exceptions depending on a template argument, needs to explicitly specify its behavior using noexcept(<condition>) specifier.

Note that it is assumed that a function which does not contain explicit noexcept specification throws exceptions, similarly to functions that declare noexcept(false) specifier.

Example

```
1  //% $Id: A15-4-4.cpp 289436 2017-10-04 10:45:23Z michal.szczepankiewicz $
2  #include <iostream>
3  #include <stdexcept>
4  void F1();  // Compliant - f1, without noexcept specification, declares to throw
5              // exceptions implicitly
6  void F2() noexcept;          // Compliant - f2 does not throw exceptions
7  void F3() noexcept(true);    // Compliant - f3 does not throw exceptions
8  void F4() noexcept(false);   // Compliant - f4 declares to throw exceptions
9  void F5() noexcept           // Compliant - f5 does not throw exceptions
10 {
11     try
12     {
```

```
13          F1();  // Exception handling needed, f1 has no noexcept specification
14      }
15
16      catch (std::exception& e)
17      {
18          // Handle exceptions
19      }
20
21      F2();  // Exception handling not needed, f2 is noexcept
22      F3();  // Exception handling not needed, f3 is noexcept(true)
23
24      try
25      {
26          F4();  // Exception handling needed, f4 is noexcept(false)
27      }
28
29      catch (std::exception& e)
30      {
31          // Handle exceptions
32      }
33  }
34  template <class T>
35  void F6() noexcept(noexcept(T()));  // Compliant - function f6() may be
36                                      // noexcept(true) or noexcept(false)
37                                      // depending on constructor of class T
38  template <class T>
39  class A
40  {
41    public:
42      A() noexcept(noexcept(T()))  // Compliant - constructor of class A may be
43      // noexcept(true) or noexcept(false) depending on
44      // constructor of class T
45      {
46      }
47  };
48  class C1
49  {
50    public:
51      C1()
52      noexcept(
53          true)  // Compliant - constructor of class C1 does not throw exceptions
54      {
55      }
56      // ...
57  };
58  class C2
59  {
60    public:
61      C2()  // Compliant - constructor of class C2 throws exceptions
62      {
63      }
```

```
64      // ...
65  };
66  void F7() noexcept  // Compliant - f7 does not throw exceptions
67  {
68      std::cout << noexcept(A<C1>()) << '\n';  // prints '1' - constructor of
69                                               // A<C1> class is noexcept(true)
70                                               // because constructor of C1 class
71                                               // is declared to be noexcept(true)
72      std::cout << noexcept(A<C2>()) << '\n';  // prints '0' - constructor of
73                                               // A<C2> class is noexcept(false)
74                                               // because constructor of C2 class
75                                               // has no noexcept specifier
76  }
```

See also

- none

Rule A15-4-5 (required, implementation, automated)
Checked exceptions that could be thrown from a function shall be specified together with the function declaration and they shall be identical in all function declarations and for all its overriders.

Rationale

In C++ language, all exceptions are unchecked, because the compiler does not force to either handle the exception or specify it. Because dynamic-exception specification is obsolete and error prone, an alternative mechanism of specifying checked exceptions using C++ comments along with function declarations is used. It is a concept that is based on Java exception handling mechanism.

When analyzing a given function f, a static code analysis needs to analyze functions invoked by f and analyze if they throw any checked exceptions that are not caught by f and not listed by f in the function comment.

Exception

Within generic code, it is not generally possible to know what types of exceptions may be thrown from operations on template arguments, so a precise exception specification cannot be written. Therefore, this rule does not apply for templates.

Example

```
1  //% $Id: A15-4-5.cpp 309502 2018-02-28 09:17:39Z michal.szczepankiewicz $
2  #include <cstdint>
3  #include <stdexcept>
4
5  /// @checkedException
6  class CommunicationError : public std::exception
7  {
```

```
8       // Implementation
9   };
10  /// @checkedException
11  class BusError : public CommunicationError
12  {
13      // Implementation
14  };
15  /// @checkedException
16  class Timeout : public std::runtime_error
17  {
18    public:
19      using std::runtime_error::runtime_error;
20      // Implementation
21  };
22  /// @throw CommunicationError  Communication error
23  /// @throw BusError  Bus error
24  /// @throw Timeout   On send timeout exception
25  void Send1(
26      std::uint8_t* buffer,
27      std::uint8_t bufferLength) noexcept(false)  // Compliant - All and only
28                                                  // those checked exceptions
29                                                  // that can be thrown are
30                                                  // specified
31  {
32      // ...
33      throw CommunicationError();
34      // ...
35      throw BusError();
36      // ...
37      throw Timeout("Timeout reached");
38      // ...
39  }
40  /// @throw CommunicationError  Communication error
41  void Send2(
42      std::uint8_t* buffer,
43      std::uint8_t bufferLength) noexcept(false)  // Non-compliant - checked
44                                                  // exceptions that can be
45                                                  // thrown are missing from
46                                                  // specification
47  {
48      // ...
49      throw CommunicationError();
50      // ...
51      throw Timeout("Timeout reached");
52      // ...
53  }
54  class MemoryPartitioningError : std::exception
55  {
56      // Implementation
57  };
58  /// @throw CommunicationError  Communication error
```

```
59 /// @throw BusError  Bus error
60 /// @throw Timeout   On send timeout exception
61 /// @throw MemoryPartitioningError  Memory partitioning error prevents message
62 /// from being sent.
63 void Send3(
64     std::uint8_t* buffer,
65     std::uint8_t bufferLength) noexcept(false)  // Non-compliant - additional
66                                                 // checked exceptions are
67                                                 // specified
68 {
69     // ...
70     throw CommunicationError();
71     // ...
72     throw Timeout("Timeout reached");
73     // ...
74 }
```

See also

- Effective Java 2nd Edition [15]: Item 62: Document all exceptions thrown by each method

6.15.5 Special functions

Rule A15-5-1 (required, implementation, automated)
All user-provided class destructors, deallocation functions, move constructors, move assignment operators and swap functions shall not exit with an exception. A noexcept exception specification shall be added to these functions as appropriate.

Rationale

When an exception is thrown, the call stack is unwound up to the point where the exception is to be handled. The destructors for all automatic objects declared between the point where the exception is thrown and where it is to be handled will be invoked. If one of these destructors or delete operators exits with an exception, then the program will terminate in an implementation-defined manner.

Move constructors and move assignment operators are usually expected to be non-throwing. If they throw exceptions, strong exception safety cannot be guaranteed, because the original type values could be already modified or partially modified.

Note that some operations in the standard library statically check the noexcept specification of the move constructors and move assignment operators of parameter types. They may choose less efficient algorithms or provide fewer exception safety guarantees if these are not noexcept.

The standard-library containers and algorithms will not work correctly if swapping of two elements exits with an exception. A non-throwing swap function is also an important basic tool to implement the strong exception safety guarantee in a copy assignment operator (see A12-8-2).

Note that it is acceptable for a destructor or deallocation function to throw an exception that is handled within this function, for example within a try-catch block.

Note that deallocation functions are declared noexcept by default. A destructor is declared as noexcept by default unless a destructor of any base class or member is potentially-throwing. Using a base class or member with a potentially-throwing destructor is a violation of this rule. The respective base class or member destructor must be fixed in order to comply to this rule. The intention of this rule is that the implementation of a user-provided destructor is ensured to not exit with an exception. Only then, the default noexcept specification added implicitly to the user-provided destructor is correct. It may be explicitly restated as noexcept for documentation purposes.

The compiler also adds a noexcept specification implicitly for any defaulted special member function. This noexcept specification depends on the noexcept specification of the member and base class operations that the defaulted special member function will call implicitly. It is therefore not required to default a special member function *only* to add the noexcept specification. Reasons to default a special member function exist independently from this rule, for example due to A12-0-1.

Exception

Generic move constructors, generic move assignment operators, and generic swap functions may have noexcept specifications which depend on type properties of the template parameters.

Example

```
1  //% $Id: A15-5-1.cpp 309720 2018-03-01 14:05:17Z jan.babst $
2  #include <stdexcept>
3  #include <type_traits>
4
5  class C1
6  {
7    public:
8      C1() = default;
9
10     // Compliant - move constructor is non-throwing and declared to be noexcept
11     C1(C1&& rhs) noexcept {}
12
13     // Compliant - move assignment operator is non-throwing and declared to be
14     // noexcept
15     C1& operator=(C1&& rhs) noexcept { return *this; }
16
17     // Compliant - destructor is non-throwing and declared to be noexcept by
18     // default
```

```
19      ~C1() noexcept {}
20  };
21
22  void Swap(C1& lhs,
23            C1& rhs) noexcept  // Compliant - swap function is non-throwing and
24                               // declared to be noexcept
25  {
26      // Implementation
27  }
28
29  class C2
30  {
31    public:
32      C2() = default;
33
34      // Compliant - move constructor is non-throwing and declared to be noexcept
35      C2(C2&& rhs) noexcept
36      {
37          try
38          {
39              // ...
40              throw std::runtime_error(
41                  "Error");  // Exception will not escape this function
42          }
43
44          catch (std::exception& e)
45          {
46              // Handle error
47          }
48      }
49
50      C2& operator=(C2&& rhs) noexcept
51      {
52          try
53          {
54              // ...
55              throw std::runtime_error(
56                  "Error");  // Exception will not escape this function
57          }
58
59          catch (std::exception& e)
60          {
61              // Handle error
62          }
63          return *this;
64      }
65
66      // Compliant - destructor is non-throwing and declared to be noexcept by
67      // default
68      ~C2()
69      {
```

```
70          try
71          {
72              // ...
73              throw std::runtime_error(
74                  "Error");  // Exception will not escape this function
75          }
76
77          catch (std::exception& e)
78          {
79              // Handle error
80          }
81      }
82  };
83
84  // Non-compliant - swap function is declared to be noexcept(false)
85  void Swap(C2& lhs, C2& rhs) noexcept(false)
86  {
87      // ...
88      // Non-compliant - Implementation exits with an exception
89      throw std::runtime_error("Swap function failed");
90  }
91
92  class C3
93  {
94    public:
95      C3() = default;
96      C3(C3&& rhs)  // Non-compliant - move constructor throws
97      {
98          // ...
99          throw std::runtime_error("Error");
100     }
101     C3& operator=(C3&& rhs)  // Non-compliant - move assignment operator throws
102     {
103         // ...
104         throw std::runtime_error("Error");
105         return *this;
106     }
107     ~C3()  // Non-compliant - destructor exits with an exception
108     {
109         throw std::runtime_error("Error");
110     }
111     static void operator delete(void* ptr, std::size_t sz)
112     {
113         // ...
114         throw std::runtime_error("Error");  // Non-compliant - deallocation
115                                             // function exits with an exception
116     }
117 };
118
119 void Fn()
120 {
```

```
    C3 c1;  // program terminates when c1 is destroyed
    C3* c2 = new C3;
    // ...
    delete c2;  // program terminates when c2 is deleted
}

template <typename T>
class Optional
{
  public:
    // ...

    // Compliant by exception
    Optional(Optional&& other) noexcept(
        std::is_nothrow_move_constructible<T>::value)
    {
        // ...
    }

    // Compliant by exception
    Optional& operator=(Optional&& other) noexcept(
        std::is_nothrow_move_assignable<T>::value&&
            std::is_nothrow_move_constructible<T>::value)
    {
        // ...
        return *this;
    }

    // ...
};
```

See also

- MISRA C++ 2008 [7]: 15-5-1: A class destructor shall not exit with an exception.
- HIC++ v4.0 [9]: 15.2.1: Do not throw an exception from a destructor
- C++ Core Guidelines [11]: E.16: Destructors, deallocation, and swap must never fail
- C++ Core Guidelines [11]: C.85: Make swap noexcept
- ISO/IEC 14882:2014 [3]: 15.4: [except.spec]
- ISO/IEC 14882:2014 [3]: 20.2.4, paragraph 9: [forward]
- A12-0-1 in section 6.12.0
- A12-8-2 in section 6.12.8

Rule A15-5-2 (required, implementation, partially automated)
Program shall not be abruptly terminated. In particular, an implicit or

explicit invocation of std::abort(), std::quick_exit(), std::_Exit(), std::terminate() shall not be done.

Rationale

Functions that are used to terminate the program in an immediate fashion, i.e. std::abort(), std::quick_exit(), std::_Exit(), do so without calling exit handlers or calling destructors of automatic, thread or static storage duration objects. It is implementation-defined whether opened streams are flushed and closed, and temporary files are removed.

The std::terminate() function calls std::abort() implicitly in its terminate handler, and it is implementation-defined whether or not stack unwinding will occur.

Note: std::terminate_handler shall not be used.

Example

```
1  //% $Id: A15-5-2.cpp 289436 2017-10-04 10:45:23Z michal.szczepankiewicz $
2  #include <cstdlib>
3  #include <exception>
4  void F1() noexcept(false);
5  void F2()  // Non-compliant
6  {
7      F1();  // A call to throwing f1() may result in an implicit call to
8             // std::terminate()
9  }
10 void F3()  // Compliant
11 {
12     try
13     {
14         F1();  // Handles all exceptions from f1() and does not re-throw
15     }
16     catch (...)
17     {
18         // Handle an exception
19     }
20 }
21 void F4(const char* log)
22 {
23     // Report a log error
24     // ...
25     std::exit(0);  // Call std::exit() function which safely cleans up resources
26 }
27 void F5()  // Compliant by exception
28 {
29     try
30     {
31         F1();
32     }
33     catch (...)
34     {
```

```
35          F4("f1() function failed");
36      }
37  }
38  int main(int, char**)
39  {
40      if (std::atexit(&F2) != 0)
41      {
42          // Handle an error
43      }
44
45      if (std::atexit(&F3) != 0)
46      {
47          // Handle an error
48      }
49
50      // ...
51      return 0;
52  }
```

See also

- MISRA C++ 2008 [7]: 15-5-3 (Required) The terminate() function shall not be called implicitly.
- HIC++ v4.0 [9]: 15.3.2 Ensure that a program does not result in a call to std::terminate
- SEI CERT C++ [10]: ERR50-CPP. Do not abruptly terminate the program

Rule A15-5-3 (required, implementation, automated)
The std::terminate() function shall not be called implicitly.

Rationale

It is implementation-defined whether the call stack is unwound before std::terminate() is called. There is no guarantee that the destructors of automatic thread or static storage duration objects will be called.

These are following ways to call std::terminate() function implicitly, according to (std::terminate() in CppReference [16]):

1. an exception is thrown and not caught (it is implementation-defined whether any stack unwinding is done in this case)
2. an exception is thrown during exception handling (e.g. from a destructor of some local object, or from a function that had to be called during exception handling)
3. the constructor or the destructor of a static or thread-local object throws an exception
4. a function registered with std::atexit or std::at_quick_exit throws an exception

5. a noexcept specification is violated (it is implementation-defined whether any stack unwinding is done in this case)
6. a dynamic exception specification is violated and the default handler for std::unexpected is executed
7. a non-default handler for std::unexpected throws an exception that violates the previously violated dynamic exception specification, if the specification does not include std::bad_exception
8. std::nested_exception::rethrow_nested is called for an object that isn't holding a captured exception
9. an exception is thrown from the initial function of std::thread
10. a joinable std::thread is destroyed or assigned to

Note: std::terminate_handler shall not be used.

Example

```
1  //% $Id: A15-5-3.cpp 289436 2017-10-04 10:45:23Z michal.szczepankiewicz $
2  #include <stdexcept>
3  #include <thread>
4  extern bool F1();
5  class A
6  {
7    public:
8      A() noexcept(false)
9      {
10         // ...
11         throw std::runtime_error("Error1");
12     }
13     ~A()
14     {
15         // ...
16         throw std::runtime_error("Error2");  // Non-compliant - std::terminate()
17                                              // called on throwing an exception
18                                              // from noexcept(true) destructor
19     }
20 };
21 class B
22 {
23   public:
24     ~B() noexcept(false)
25     {
26         // ...
27         throw std::runtime_error("Error3");
28     }
29 };
30 void F2()
31 {
32     throw;
```

```
}
void ThreadFunc()
{
    A a;  // Throws an exception from a's constructor and does not handle it in
          // thread_func()
}
void F3()
{
    try
    {
        std::thread t(&ThreadFunc);  // Non-compliant - std::terminate() called
                                     // on throwing an exception from
                                     // thread_func()

        if (F1())
        {
            throw std::logic_error("Error4");
        }

        else
        {
            F2();  // Non-compliant - std::terminate() called if there is no
                   // active exception to be re-thrown by f2
        }
    }
    catch (...)
    {
        B b;  // Non-compliant - std::terminate() called on throwing an
              // exception from b's destructor during exception handling

        // ...
        F2();
    }
}
static A a;  // Non-compliant - std::terminate() called on throwing an exception
             // during program's start-up phase
int main(int, char**)
{
    F3();  // Non-compliant - std::terminate() called if std::logic_error is
           // thrown
    return 0;
}
```

See also

- MISRA C++ 2008 [7]: 15-5-3 (Required) The terminate() function shall not be called implicitly.

6.16 Preprocessing directives

6.16.0 General

Rule A16-0-1 (required, implementation, automated)
The pre-processor shall only be used for unconditional and conditional file inclusion and include guards, and using the following directives: (1) #ifndef, (2) #ifdef, (3) #if, (4) #if defined, (5) #elif, (6) #else, (7) #define, (8) #endif, (9) #include.

Rationale

C++ provides safer, more readable and easier to maintain ways of achieving what is often done using the pre-processor. The pre-processor does not obey the linkage, lookup and function call semantics. Instead, constant objects, constexprs, inline functions and templates are to be used.

Example

```
// $Id: A16-0-1.cpp 289436 2017-10-04 10:45:23Z michal.szczepankiewicz $
#pragma once  // Non-compliant - implementation-defined feature

#ifndef HEADER_FILE_NAME  // Compliant - include guard
#define HEADER_FILE_NAME  // Compliant - include guard

#include <cstdint>  // Compliant - unconditional file inclusion

#ifdef WIN32
#include <windows.h>  // Compliant - conditional file inclusion
#endif

#ifdef WIN32
std::int32_t fn1(
    std::int16_t x,
    std::int16_t y) noexcept;  // Non-compliant - not a file inclusion
#endif

#if defined VERSION && VERSION > 2011L  // Compliant
#include <array>       // Compliant - conditional file inclusion
#elif VERSION > 1998L  // Compliant
#include <vector>      // Compliant - conditional file inclusion
#endif                 // Compliant

#define MAX_ARRAY_SIZE 1024U  // Non-compliant
#ifndef MAX_ARRAY_SIZE        // Non-compliant
#error "MAX_ARRAY_SIZE has not been defined"  // Non-compliant
#endif                 // Non-compliant
#undef MAX_ARRAY_SIZE  // Non-compliant

#define MIN(a, b) (((a) < (b)) ? (a) : (b))  // Non-compliant
```

```
#define PLUS2(X) ((X) + 2)  // Non-compliant - function should be used instead
#define PI 3.14159F         // Non-compliant - constexpr should be used instead
#define std ::int32_t long  // Non-compliant - 'using' should be used instead
#define STARTIF if( // Non-compliant - language redefinition
#define HEADER "filename.h"  // Non-compliant - string literal

void Fn2() noexcept
{
#ifdef __linux__  // Non-compliant - ifdef not used for file inclusion

// ...

#elif WIN32  // Non-compliant - elif not used for file inclusion

// ...

#elif __APPLE__  // Non-compliant - elif not used for file inclusion

// ...

#else  // Non-compliant - else not used for file inclusion

// ...

#endif  // Non-compliant - endif not used for file inclusion or include guards
}

#endif  // Compliant - include guard
```

See also

- MISRA C++ 2008 [7]: Rule 16-2-1 The pre-processor shall only be used for file inclusion and include guards.
- MISRA C++ 2008 [7]: Rule 16-2-2 C++ macros shall only be used for: include guards, type qualifiers, or storage class specifiers.
- JSF December 2005 [8]: AV Rule 26 Only the following pre-processor directives shall be used: 1. #ifndef 2. #define 3. #endif 4. #include.
- JSF December 2005 [8]: AV Rule 27 #ifndef, #define and #endif will be used to prevent multiple inclusions of the same header file. Other techniques to prevent the multiple inclusions of header files will not be used.
- JSF December 2005 [8]: AV Rule 28 The #ifndef and #endif pre-processor directives will only be used as defined in AV Rule 27 to prevent multiple inclusions of the same header file.
- JSF December 2005 [8]: AV Rule 29 The #define pre-processor directive shall not be used to create inline macros. Inline functions shall be used instead.

- JSF December 2005 [8]: AV Rule 30 The #define pre-processor directive shall not be used to define constant values. Instead, the const qualifier shall be applied to variable declarations to specify constant values.
- JSF December 2005 [8]: AV Rule 31 The #define pre-processor directive will only be used as part of the technique to prevent multiple inclusions of the same header file.
- JSF December 2005 [8]: AV Rule 32 The #include pre-processor directive will only be used to include header (*.h) files.
- HIC++ v4.0 [9]: 16.1.1 Use the preprocessor only for implementing include guards, and including header files with include guards.

Rule M16-0-1 (required, implementation, automated)
#include directives in a file shall only be preceded by other pre-processor directives or comments.

See MISRA C++ 2008 [7]

Rule M16-0-2 (required, implementation, automated)
Macros shall only be #define'd or #undef'd in the global namespace.

See MISRA C++ 2008 [7]

Rule M16-0-5 (required, implementation, automated)
Arguments to a function-like macro shall not contain tokens that look like pre-processing directives.

See MISRA C++ 2008 [7]

Note: Function-like macros are anyway not allowed, see A16-0-1. This rule is kept in case A16-0-1 is disabled in a project.

Rule M16-0-6 (required, implementation, automated)
In the definition of a function-like macro, each instance of a parameter shall be enclosed in parentheses, unless it is used as the operand of # or ##.

See MISRA C++ 2008 [7]

Note: Function-like macros are anyway not allowed, see A16-0-1. This rule is kept in case A16-0-1 is disabled in a project.

Rule M16-0-7 (required, implementation, automated)
Undefined macro identifiers shall not be used in #if or #elif pre-processor directives, except as operands to the defined operator.

See MISRA C++ 2008 [7]

Note: “#if” and “#elif” are anyway only allowed for conditional file inclusion, see A16-0-1. This rule is kept in case A16-0-1 is disabled in a project.

Rule M16-0-8 (required, implementation, automated)
If the # token appears as the first token on a line, then it shall be immediately followed by a pre-processing token.

See MISRA C++ 2008 [7]

6.16.1 Conditional inclusion

Rule M16-1-1 (required, implementation, automated)
The defined pre-processor operator shall only be used in one of the two standard forms.

See MISRA C++ 2008 [7]

Note: “#if defined” is anyway only allowed for conditional file inclusion, see A16-0-1. This rule is kept in case A16-0-1 is disabled in a project.

Rule M16-1-2 (required, implementation, automated)
All #else, #elif and #endif pre-processor directives shall reside in the same file as the #if or #ifdef directive to which they are related.

See MISRA C++ 2008 [7]

Note: “#if”, “#elif”, “#else” and “#ifded” are anyway only allowed for conditional file inclusion, see A16-0-1. This rule is kept in case A16-0-1 is disabled in a project.

6.16.2 Source file inclusion

Rule M16-2-3 (required, implementation, automated)
Include guards shall be provided.

See MISRA C++ 2008 [7]

Rule A16-2-1 (required, implementation, automated)
The ', ", /*, //, \ characters shall not occur in a header file name or in #include directive.

Rationale

It is undefined behavior if the ', ", /*, //, \\ characters are used in #include directive, between < and > or " " delimiters.

Example

```
// $Id: A16-2-1.cpp 271687 2017-03-23 08:57:35Z piotr.tanski $

// #include <directory/headerfile.hpp>  // Compliant
// #include <headerfile.hpp>  // Compliant
// #include "directory/headerfile.hpp"  // Compliant
// #include "headerfile.hpp"  // Compliant
// #include <directory/*.hpp>  // Non-compliant
// #include <header'file.hpp>  // Non-compliant
// #include <"headerfile.hpp">  // Non-compliant
// #include <directory\\headerfile.hpp>  // Non-compliant
```

See also

- MISRA C++ 2008 [7]: Rule 16-2-4 The ', ", /* or // characters shall not occur in a header file name.
- MISRA C++ 2008 [7]: Rule 16-2-5 The \character shall not occur in a header file name.

Rule A16-2-2 (required, implementation, automated)
There shall be no unused include directives.

Rationale

Presence of unused include directives considerably slows down compilation phase, makes the code base larger and introduces unneeded dependencies.

Note: In order to determine what an unused include directive is, only the immediate level of includes, and the specifications of external libraries shall be considered. So, for example, if a source file uses the standard library algorithm `std::copy`, it is required (see also rule A16-2-3) to include the standard library header `<algorithm>`. It is not a violation of this rule if `<algorithm>`, possibly through inclusion of other headers, contains declarations of symbols not used in the source file.

Example

```
// $Id: A16-2-2.cpp 289436 2017-10-04 10:45:23Z michal.szczepankiewicz $
#include <algorithm>  // Non-compliant - nothing from algorithm header file is
    used
```

```
#include <array>      // Non-compliant - nothing from array header file is used
#include <cstdint>    // Compliant - std::int32_t, std::uint8_t are used
#include <iostream>   // Compliant - cout is used
#include <stdexcept>  // Compliant - out_of_range is used
#include <vector>     // Compliant - vector is used
void Fn1() noexcept
{
    std::int32_t x = 0;
    // ...
    std::uint8_t y = 0;
    // ...
}
void Fn2() noexcept(false)
{
    try
    {
        std::vector<std::int32_t> v;
        // ...
        std::uint8_t idx = 3;
        std::int32_t value = v.at(idx);
    }
    catch (std::out_of_range& e)
    {
        std::cout << e.what() << '\n';
    }
}
```

See also

- HIC++ v4.0 [9]: 16.1.5 Include directly the minimum number of headers required for compilation.

Rule A16-2-3 (required, implementation, non-automated)
An include directive shall be added explicitly for every symbol used in a file.

Rationale

All header files that define types or functions used in a file should be included explicitly. The actual header to include depends on the specification of the library/component used.

Exception

Types defined via forward declarations do not violate this rule.

Example

```
// $Id: A16-2-3.hpp 319944 2018-05-21 09:00:40Z ilya.burylov $
#ifndef HEADER_HPP
```

```
#define HEADER_HPP

#include <array>
#include <cstdint>

class B;  // Compliant - type B can be included using forward declaration

class OutOfRangeException
    : public std::out_of_range  // Non-compliant - <stdexcept> which defines
                                // out_of_range included
                                // implicitly through <array>
{
  public:
    using std::out_of_range::out_of_range;
};

class A
{
  public:
    // Interface of class A

  private:
    std::array<std::uint32_t, 10>
        mArray;  // Compliant - <array> included explicitly
    B* mB;
    std::int32_t mX;  // Compliant - <cstdint> included explicitly
};

#endif
```

See also

- none

6.16.3 Macro replacement

Rule M16-3-1 (required, implementation, automated)
There shall be at most one occurrence of the # or ## operators in a single macro definition.

See MISRA C++ 2008 [7]

Note: Operators # and ## are anyway not allowed, see M16-3-2. This rule is kept in case M16-3-2 is disabled in a project.

Rule M16-3-2 (advisory, implementation, automated)
The # and ## operators should not be used.

See MISRA C++ 2008 [7]

6.16.6 Error directive

Rule A16-6-1 (required, implementation, automated)
#error directive shall not be used.

Rationale

Using the pre-processor #error directive may lead to code that is complicated and not clear for developers. The #error directive can not be applied to templates as it will not be evaluated as a per-instance template deduction.

Static assertion, similarly to #error directive, provides a compile-time error checking. However, static_assert behaves correctly in all C++ concepts and makes the code more readable and does not rely on pre-processor directives.

Note: "#error" is anyway not allowed, see A16-0-1. This rule is kept in case A16-0-1 is disabled in a project.

Example

```
// $Id: A16-6-1.cpp 289436 2017-10-04 10:45:23Z michal.szczepankiewicz $
#include <cstdint>
#include <type_traits>
constexpr std::int32_t value = 0;
#if value > 10
#error "Incorrect value"  // Non-compliant
#endif
void F1() noexcept
{
    static_assert(value <= 10, "Incorrect value");  // Compliant
    // ...
}
template <typename T>
void F2(T& a)
{
    static_assert(std::is_copy_constructible<T>::value,
                  "f2() function requires copying");  // Compliant
    // ...
}
```

See also

- none

6.16.7 Pragma directive

Rule A16-7-1 (required, implementation, automated)
The #pragma directive shall not be used.

Rationale

The #pragma directive is implementation-defined and causes the implementation to behave in implementation-defined manner.

Example

```
1 // $Id: A16-7-1.hpp 270497 2017-03-14 14:58:50Z piotr.tanski $
2 // #pragma once  // Non-compliant - implementation-defined manner
3 #ifndef A16_7_1_HPP  // Compliant - equivalent to #pragma once directive
4 #define A16_7_1_HPP
5
6 // ...
7
8 #endif
```

See also

- MISRA C++ 2008 [7]: Rule 16-6-1 All uses of the #pragma directive shall be documented.

6.17 Library introduction - partial

6.17.1 General

Rule A17-0-1 (required, implementation, automated)
Reserved identifiers, macros and functions in the C++ standard library shall not be defined, redefined or undefined.

Rationale

It is generally bad practice to #undef a macro that is defined in the standard library. It is also bad practice to #define a macro name that is a C++ reserved identifier, or C++ keyword or the name of any macro, object or function in the standard library. For example, there are some specific reserved words and function names that are known to give rise to undefined behavior if they are redefined or undefined, including defined, __LINE__, __FILE__, __DATE__, __TIME__, __STDC__, errno and assert.

Refer to C++ Language Standard for a list of the identifiers that are reserved. Generally, all identifiers that begin with the underscore character are reserved.

Note that this rule applies regardless of which header files, if any, are actually included.

Example

```
1 // $Id: A17-0-1.cpp 271389 2017-03-21 14:41:05Z piotr.tanski $
2 #undef __TIME__       // Non-compliant
3 #define __LINE__ 20   // Non-compliant
```

See also

- MISRA C++ 2008 [7]: Rule 17-0-1 Reserved identifiers, macros and functions in the standard library shall not be defined, redefined or undefined.

Rule M17-0-2 (required, implementation, automated)
The names of standard library macros and objects shall not be reused.

See MISRA C++ 2008 [7]

Rule M17-0-3 (required, implementation, automated)
The names of standard library functions shall not be overridden.

See MISRA C++ 2008 [7]

Rule A17-0-2 (required, implementation, non-automated)
All project's code including used libraries (including standard and user-defined libraries) and any third-party user code shall conform to the AUTOSAR C++14 Coding Guidelines.

Rationale

Note that library code can be provided as source code or be provided in a compiled form. The rule applies for any form of libraries.

As for any rule in this standard, a deviation procedure can be performed for this rule and the project needs to argue what are the measures ensuring that non-compliant libraries can be used in a project, addressing:

1. interference from the non-compliant code (for example, a library function overwrites the stack or heap of the caller)
2. residual errors in non-compliant code resulting with its wrong outputs, which are subsequently used (for example, a library function delivers wrong return value used by the caller).

Exception

If a function is defined in a library or any third-party user code but it is ensured that the function will not be used (directly or indirectly) in the project, then the function may not conform to the AUTOSAR C++14 Coding Guidelines.

See also

- none

Rule M17-0-5 (required, implementation, automated)
The setjmp macro and the longjmp function shall not be used.

See MISRA C++ 2008 [7]

See: A6-6-1.

6.17.2 The C standard library

Rule A17-1-1 (required, implementation, non-automated)
Use of the C Standard Library shall be encapsulated and isolated.

Rationale

The C Standard Library leaves the responsibility for handling errors, data races and security concerns up to developers. Therefore, use of the C Standard Library needs to be separated and wrapped with functions that will be fully responsible for all specific checks and error handling.

Example

```
1  // $Id: A17-1-1.cpp 289436 2017-10-04 10:45:23Z michal.szczepankiewicz $
2  #include <cerrno>
3  #include <cstdio>
4  #include <cstring>
5  #include <iostream>
6  #include <stdexcept>
7
8  void Fn1(const char* filename)  // Compliant - C code is isolated; fn1()
9                                  // function is a wrapper.
10 {
11     FILE* handle = fopen(filename, "rb");
12     if (handle == NULL)
13     {
14         throw std::system_error(errno, std::system_category());
15     }
16     // ...
17     fclose(handle);
```

```
}

void Fn2() noexcept
{
    try
    {
        Fn1("filename.txt");  // Compliant - fn1() allows you to use C code like
                              // C++ code

        // ...
    }
    catch (std::system_error& e)
    {
        std::cerr << "Error: " << e.code() << " - " << e.what() << '\n';
    }
}

std::int32_t Fn3(const char* filename) noexcept  // Non-compliant - placing C
// functions calls along with C++
// code forces a developer to be
// responsible for C-specific error
// handling, explicit resource
// cleanup, etc.
{
    FILE* handle = fopen(filename, "rb");
    if (handle == NULL)
    {
        std::cerr << "An error occured: " << errno << " - " << strerror(errno)
                  << '\n';
        return errno;
    }

    try
    {
        // ...
        fclose(handle);
    }
    catch (std::system_error& e)
    {
        fclose(handle);
    }
    catch (std::exception& e)
    {
        fclose(handle);
    }

    return errno;
}
```

See also

- MISRA C++ 2008 [7]: Rule 19-3-1 The error indicator errno shall not be used.

- HIC++ v4.0 [9]: 17.2.1 Wrap use of the C Standard Library.
- JSF December 2005 [8]: Chapter 4.5.1: Standard Libraries, AV Rule 17 - AV Rule 25.

6.17.3 Definitions

The corresponding section in the C++14 standard provides a glossary only.

6.17.6 Library-wide requirements

Rule A17-6-1 (required, implementation, automated)
Non-standard entities shall not be added to standard namespaces.

Rationale

Adding declarations or definitions to namespace `std` or its sub-namespaces, or to namespace `posix` or its sub-namespaces leads to undefined behavior.

Declaring an explicit specialization of a member function or member function template of a standard library class or class template leads to undefined behavior.

Declaring an explicit or partial specialization of a member class template of a standard library class or class template leads to undefined behavior.

Exception

It is allowed by the language standard to add a specialization to namespace `std` if the declaration depends on a user-defined type, meets the requirements for the original template and is not explicitly forbidden.

It is allowed by the language standard to explicitly instantiate a template defined in the standard library if the declaration depends on a user defined type and meets the requirements for the original template.

Example

```
1  // $Id: A17-6-1.cpp 305588 2018-01-29 11:07:35Z michal.szczepankiewicz $
2  #include <cstdint>
3  #include <limits>
4  #include <memory>
5  #include <type_traits>
6  #include <utility>
7
8  namespace std
9  {
10
11 // Non-compliant - An alias definition is added to namespace std.
```

```
// This is a compile error in C++17, since std::byte is already defined.
using byte = std::uint8_t;

// Non-compliant - A function definition added to namespace std.
pair<int, int> operator+(pair<int, int> const& x, pair<int, int> const& y)
{
    return pair<int, int>(x.first + y.first, x.second + y.second);
}

}  // namespace std

struct MyType
{
    int value;
};

namespace std
{

// Non-compliant - std::numeric_limits may not be specialized for
// non-arithmetic types [limits.numeric].
template <>
struct numeric_limits<MyType> : numeric_limits<int>
{
};

// Non-compliant - Structures in <type_traits>, except for std::common_type,
// may not be specialized [meta.type.synop].
template <>
struct is_arithmetic<MyType> : true_type
{
};

// Compliant - std::hash may be specialized for a user type if the
// specialization fulfills the requirements in [unord.hash].
template <>
struct hash<MyType>
{
    using result_type = size_t;    // deprecated in C++17
    using argument_type = MyType;  // deprecated in C++17

    size_t operator()(MyType const& x) const noexcept
    {
        return hash<int>()(x.value);
    }
};

}  // namespace std
```

See also

- SEI CERT C++ Coding Standard [10]: DCL58-CPP: Do not modify the standard namespaces
- C++ Core Guidelines [11]: SL.3: Do not add non-standard entities to namespace std
- ISO/IEC 14882:2014 [3]: 17.6.4.2: [namespace.constraints]
- ISO/IEC 14882:2014 [3]: 18.3.2.1: [limits.numeric]
- ISO/IEC 14882:2014 [3]: 20.9.13: [unord.hash]
- ISO/IEC 14882:2014 [3]: 20.10.2: [meta.type.synop]

6.18 Language support library - partial

The corresponding chapter in the C++ standard defines the fundamental support libraries, including integer types, dynamic memory, start and termination.

6.18.0 General

Rule A18-0-1 (required, implementation, automated)
The C library facilities shall only be accessed through C++ library headers.

Rationale

C libraries (e.g. <stdio.h>) also have corresponding C++ libraries (e.g. <cstdio>). This rule requires that the C++ version is used.

See also

- MISRA C++ 2008 [7]: Rule 18-0-1 (Required) The C library shall not be used.
- HIC++ v4.0 [9]: 1.3.3 Do not use the C Standard Library .h headers.

Rule A18-0-2 (required, implementation, automated)
The error state of a conversion from string to a numeric value shall be checked.

Rationale

Error conditions of a string-to-number conversion must be detected and properly handled. Such errors can happen when an input string:

- does not contain a number;
- contains a number, but it is out of range;

- contains additional data after a number.

Some functions for string-to-number conversion from the C Standard library have undefined behavior when a string cannot be converted to a number, e.g. `atoi()`. Since the C++11 Language Standard, new numeric conversion functions are available (see: `std::stoi()`, `std::stol()`, `std::stoll()` [16]). These guarantee defined behavior.

Moreover, errors shall be checked also for formatted input stream functions (e.g. `istream::operator>>()`), by using `basic_ios::fail()`.

Example

```
// $Id: A18-0-2.cpp 312092 2018-03-16 15:47:01Z jan.babst $
#include <cstdint>
#include <cstdlib>
#include <iostream>
#include <string>

std::int32_t F1(const char* str) noexcept
{
    return atoi(str);  // Non-compliant - undefined behavior if str can not
                       // be converted
}
std::int32_t F2(std::string const& str) noexcept(false)
{
    return std::stoi(str);  // Compliant - throws a std::invalid_argument
                            // exception if str can not be converted
}

std::uint16_t ReadFromStdin1()  // non-compliant
{
    std::uint16_t a;
    std::cin >> a;  // no error detection
    return a;
}

std::uint16_t ReadFromStdin2()  // compliant
{
    std::uint16_t a;

    std::cin.clear();  // clear all flags
    std::cin >> a;
    if (std::cin.fail())
    {
        throw std::runtime_error{"unable to read an integer"};
    }
    std::cin.clear();  // clear all flags for subsequent operations
    return a;
}
```

See also

- MISRA C++ 2008 [7]: 18-0-2: The library functions atof, atoi and atol from library <cstdlib> shall not be used.
- SEI CERT C++ Coding Standard [10]: ERR34-C: Detect errors when converting a string to a number
- SEI CERT C++ Coding Standard [10]: ERR62-CPP: Detect errors when converting a string to a number

Rule M18-0-3 (required, implementation, automated)
The library functions abort, exit, getenv and system from library <cstdlib> shall not be used.

See MISRA C++ 2008 [7]

Rule M18-0-4 (required, implementation, automated)
The time handling functions of library <ctime> shall not be used.

See MISRA C++ 2008 [7]

Note: Facilities from `<chrono>` shall be used instead.

Rule M18-0-5 (required, implementation, automated)
The unbounded functions of library <cstring> shall not be used.

See MISRA C++ 2008 [7]

Note: The intention of this rule is to prohibit the functions from `<cstring>` which have a `char*` or `char const*` parameter, but no additional `size_t` parameter placing a bound on the underlying loop. Other functions from `<cstring>` taking a `char*` or `char const*` parameter fall under the restrictions of rule A27-0-4. Use of `memchr`, `memcmp`, `memset`, `memcpy`, and `memmove` is still allowed by this rule, but limited by rule A12-0-2.

See also

- Rule A12-0-2 in section 6.12.0
- Rule A27-0-4 in section 6.27.1

Rule A18-0-3 (required, implementation, automated)
The library <clocale> (locale.h) and the setlocale function shall not be used.

Rationale

A call to the setlocale function introduces a data race with other calls to setlocale function.

It may also introduce a data race with calls to functions that are affected by the current locale settings: fprintf, isprint, iswdigit, localeconv, tolower, fscanf, ispunct, iswgraph, mblen, toupper, isalnum, isspace, iswlower, mbstowcs, towlower, isalpha, isupper, iswprint, mbtowc, towupper, isblank, iswalnum, iswpunct, setlocale, wcscoll, iscntrl, iswalpha, iswspace, strcoll, wcstod, isdigit, iswblank, iswupper, strerror, wcstombs, isgraph, iswcntrl, iswxdigit, strtod, wcsxfrm, islower, iswctype, isxdigit, strxfrm, wctomb.

See also

- JSF December 2005 [8]: AV Rule 19 <locale.h> and the setlocale function shall not be used.

6.18.1 Types

Rule A18-1-1 (required, implementation, automated)
C-style arrays shall not be used.

Rationale

A C-style array is implicitly convertible to a raw pointer and easily loses information about its size. This construct is unsafe, unmaintainable, and a source of potential errors.

For fixed-size, stack-allocated arrays, `std::array` is supposed to be used instead. This type is designed to work with standard library algorithms.

Exception

It is allowed to declare a static `constexpr` data member of a C-style array type.

Example

```
1  // $Id: A18-1-1.cpp 289436 2017-10-04 10:45:23Z michal.szczepankiewicz $
2  #include <algorithm>
3  #include <array>
4  #include <cstdint>
5  void Fn() noexcept
6  {
7      const std::uint8_t size = 10;
8      std::int32_t a1[size];               // Non-compliant
9      std::array<std::int32_t, size> a2;   // Compliant
10     // ...
11     std::sort(a1, a1 + size);
12     std::sort(a2.begin(), a2.end());  // More readable and maintainable way of
13                                       // working with STL algorithms
14 }
15 class A
16 {
17   public:
```

```
18      static constexpr std::uint8_t array[]{0, 1, 2};  // Compliant by exception
19  };
```

See also

- C++ Core Guidelines [11]: ES.27: Use std::array or stack_array for arrays on the stack.
- C++ Core Guidelines [11]: SL.con.1: Prefer using STL array or vector instead of a C array.

Rule A18-1-2 (required, implementation, automated)
The std::vector<bool> specialization shall not be used.

Rationale

The `std::vector<bool>` specialization does not work with all STL algorithms as expected. In particular, `operator[]()` does not return a contiguous sequence of elements as it does for the primary template `std::vector<T>`.

The C++ Language Standard guarantees that distinct elements of an STL container can be safely modified concurrently, except when the container is a `std::vector<bool>`.

Note that `std::bitset<N>`, `std::array<bool, N>`, `std::deque<bool>`, or using `std::vector` with a value type which wraps `bool` are possible alternatives.

Example

```
1   // $Id: A18-1-2.cpp 312108 2018-03-16 17:56:49Z jan.babst $
2   #include <cstdint>
3   #include <vector>
4
5   class BoolWrapper
6   {
7     public:
8       BoolWrapper() = default;
9       constexpr BoolWrapper(bool b) : b_(b) {}        // implicit by design
10      constexpr operator bool() const { return b_; }  // implicit by design
11    private:
12      bool b_{};
13  };
14
15  void Fn() noexcept
16  {
17      std::vector<std::uint8_t> v1;                           // Compliant
18      std::vector<bool> v2;                                   // Non-compliant
19      std::vector<BoolWrapper> v3{true, false, true, false};  // Compliant
20  }
```

See also

- HIC++ v4.0 [9]: 17.1.1: Do not use std::vector<bool>.

Rule A18-1-3 (required, implementation, automated)
The std::auto_ptr type shall not be used.

Rationale

The `std::auto_ptr` type has been deprecated since the C++11 Language Standard and is removed from the C++17 Language Standard. Due to the lack of move semantics in pre C++11 standards, it provides unusual copy semantics and cannot be placed in STL containers.

The correct alternative is `std::unique_ptr`, which shall be used instead.

Example

```
1  // $Id: A18-1-3.cpp 289436 2017-10-04 10:45:23Z michal.szczepankiewicz $
2  #include <cstdint>
3  #include <memory>
4  #include <vector>
5  void Fn() noexcept
6  {
7      std::auto_ptr<std::int32_t> ptr1(new std::int32_t(10));  // Non-compliant
8      std::unique_ptr<std::int32_t> ptr2 =
9          std::make_unique<std::int32_t>(10);      // Compliant
10     std::vector<std::auto_ptr<std::int32_t>> v;  // Non-compliant
11 }
```

See also

- HIC++ v4.0 [9]: 1.3.4: Do not use deprecated STL library features.
- cppreference.com [16]: std::auto_ptr.
- A1-1-1 in section 6.1.1

Rule A18-1-4 (required, implementation, automated)
A pointer pointing to an element of an array of objects shall not be passed to a smart pointer of single object type.

Rationale

A dynamically allocated array of objects needs a corresponding deallocation function, e.g. allocation by `new[]` requires deallocation by `delete[]`, see also rule A18-5-3 in section 6.18.5. Smart pointers of a single object type, e.g. `std::unique_ptr<T>` and `std::shared_ptr<T>`, by default have a deleter associated with them which is only capable of deleting a single object. Therefore, it is undefined behavior if a pointer pointing to an element of an array of objects is passed to such a

smart pointer. With the standard library smart pointer templates `std::unique_ptr` and `std::shared_ptr`, this is possible when calling the constructor or the `reset` function.

Note that the standard library provides a specialization of the `std::unique_ptr` template for array types, `std::unique_ptr<T[]>`, and corresponding overloads for `std::make_unique`. Usage of these features is conforming to this rule.

Note that corresponding features for `std::shared_ptr` are only available in C++17 (usage of `std::shared_ptr<T[]>` with C++11 and C++14 will lead to compilation errors). The overloads for `std::make_shared` will only be available in C++20.

Furthermore, note that it is possible to create a smart pointer of single object type with a custom deleter handling an array of objects. This is well behaving as long as this smart pointer is actually managing an array of objects. However, such a use is error-prone, since the smart pointer can be assigned a single object again in the `reset` function; it may no longer be possible in C++17 (moving a `std::unique_ptr<T[]>` into a `std::shared_ptr<T>` is no longer allowed); and it is superseded by better alternatives in C++17 (availability of `std:shared_ptr<T[]>`). Therefore such usage is considered not compliant to this rule.

In many cases, using a container such as `std::array` or `std::vector` or a smart pointer to a container, e.g. `std::shared_ptr<std::vector<T>>`, is a better alternative than a smart pointer to an array of objects.

Example

```
1  // $Id: A18-1-4.cpp 313638 2018-03-26 15:34:51Z jan.babst $
2  #include <memory>
3  class A
4  {
5  };
6  void F1()
7  {
8      // Create a dynamically allocated array of 10 objects of type A.
9      auto up1 = std::make_unique<A[]>(10);  // Compliant
10
11     std::unique_ptr<A> up2{up1.release()};  // Non-compliant
12 }
13 void F2()
14 {
15     auto up1 = std::make_unique<A[]>(10);  // Compliant
16
17     std::unique_ptr<A> up2;
18     up2.reset(up1.release());  // Non-compliant
19 }
20 void F3()
21 {
22     auto up = std::make_unique<A[]>(10);  // Compliant
23
24     std::shared_ptr<A> sp{up.release()};  // Non-compliant
25 }
```

```
26 void F4()
27 {
28     auto up = std::make_unique<A[]>(10);  // Compliant
29
30     std::shared_ptr<A> sp;
31     sp.reset(up.release());  // Non-compliant
32 }
33 void F5()
34 {
35     auto up = std::make_unique<A[]>(10);  // Compliant
36
37     // sp will obtain its deleter from up, so the array will be correctly
38     // deallocated. However, this is no longer allowed in C++17.
39     std::shared_ptr<A> sp{std::move(up)};  // Non-compliant
40     sp.reset(new A{});                     // leads to undefined behavior
41 }
42 void F6()
43 {
44     auto up = std::make_unique<A[]>(10);  // Compliant
45
46     // Well behaving, but error-prone
47     std::shared_ptr<A> sp{up.release(),
48                           std::default_delete<A[]>{}};  // Non-compliant
49     sp.reset(new A{});  // leads to undefined behavior
50 }
```

See also

- HIC++ v4.0 [9]: 17.3.4: Do not create smart pointers of array type.
- ISO/IEC 14882:2014 [3]: 20.8 Smart pointers: [smartptr]
- Rule A18-5-3 in section 6.18.5

Rule A18-1-6 (required, implementation, automated)
All std::hash specializations for user-defined types shall have a noexcept function call operator.

Rationale

Some of standard library containers use std::hash indirectly. Function call operator should be defined as noexcept to prevent container simple access from throwing an exception.

Note: Consider own hash specializations to use standard library specializations combined with XOR (^) operation as implemented by `boost::hash_combine`:
$seed$ ^= std::hash$<$ $decltype$ $(v)>\{\}(v)$ + $0x9e3779b9$ + $(seed << 6)$ + $(seed >> 2)$;

Example

```
1 // $Id: A18-1-6.cpp 311792 2018-03-15 04:15:08Z christof.meerwald $
```

```
#include <cstdint>
#include <functional>
#include <string>
#include <unordered_map>

class A
{
  public:
    A(uint32_t x, uint32_t y) noexcept : x(x), y(y) {}

    uint32_t GetX() const noexcept { return x; }
    uint32_t GetY() const noexcept { return y; }

    friend bool operator == (const A &lhs, const A &rhs) noexcept
    { return lhs.x == rhs.x && lhs.y == rhs.y; }
  private:
    uint32_t x;
    uint32_t y;
};

class B
{
  public:
    B(uint32_t x, uint32_t y) noexcept : x(x), y(y) {}

    uint32_t GetX() const noexcept { return x; }
    uint32_t GetY() const noexcept { return y; }

    friend bool operator == (const B &lhs, const B &rhs) noexcept
    { return lhs.x == rhs.x && lhs.y == rhs.y; }
  private:
    uint32_t x;
    uint32_t y;
};

namespace std
{
// Compliant
template <>
struct hash<A>
{
    std::size_t operator()(const A& a) const noexcept
    {
        auto h1 = std::hash<decltype(a.GetX())>{}(a.GetX());
        std::size_t seed { h1 + 0x9e3779b9 };
        auto h2 = std::hash<decltype(a.GetY())>{}(a.GetY());
        seed ^= h2 + 0x9e3779b9 + (seed << 6) + (seed >> 2);
        return seed;
    }
};

```

```
// Non-compliant: string concatenation can potentially throw
template <>
struct hash<B>
{
    std::size_t operator()(const B& b) const
    {
        std::string s{std::to_string(b.GetX()) + ',' + std::to_string(b.GetY())};
        return std::hash<std::string>{}(s);
    }
};
}

int main()
{
    std::unordered_map<A, bool> m1 { { A{5, 7}, true } };

    if (m1.count(A{4, 3}) != 0)
    {
        // ....
    }

    std::unordered_map<B, bool> m2 { { B{5, 7}, true } };

    // Lookup can potentially throw if hash function throws
    if (m2.count(B{4, 3}) != 0)
    {
        // ....
    }
}
```

See also

- C++ Core Guidelines [11]: C.89: Make a hash noexcept.

6.18.2 Implementation properties

Rule M18-2-1 (required, implementation, automated)
The macro offsetof shall not be used.

See MISRA C++ 2008 [7]

6.18.5 Dynamic memory management

The dynamic memory management provides flexible mechanism of allocating and deallocating blocks of memory during run-time phase of the program. The application is allowed to acquire as much memory as it needs in its current state, and return it once the memory is not used.

Moreover, this is a convenient way of extending lifetime of objects outside the functions where the objects were created. In other words, a function can create objects on dynamic memory and then exit and the objects that were created in the dynamic memory are preserved and can be used subsequently by other functions.

The dynamic memory management uses the Operating System routines to allocate and deallocate memory, what introduces several issues. Therefore, the AUTOSAR C++14 Coding Guidelines defines specific rules for appropriate usage and implementation of dynamic memory management.

Challenges arising due to dynamic memory usage

Issue:	Solution:
Memory leaks	RAII design pattern usage is highly recommended for managing resource and memory acquisition and release (A18-5-2). It is prohibited to make calls to new and delete operators explicitly, to force programmers to assign each allocated memory block to manager object which deallocates the memory automatically on leaving its scope. Also, the form of delete operator used for memory deallocation needs to match the form of new operator used for memory allocation (A18-5-3).
Memory fragmentation	Memory allocator used in the project needs to guarantee that no memory fragmentation occurs (A18-5-5).
Invalid memory access	C-style functions malloc/calloc/realloc must not be used in the project, so memory block can not be accessed as it would be of another type. Memory allocator used in the project needs to guarantee that objects do not overlap in the physical storage (A18-5-5).
Erroneous memory allocations	The application program needs to define the maximum amount of dynamic memory it needs, so running out of memory must not occur during faultless execution. The memory would be pre-allocated before run-time phase of the program (A18-5-5).
Not deterministic execution time of memory allocation and deallocation	Memory allocator used in the project needs to guarantee that memory allocation and deallocation are executed within defined time constraints that are appropriate for the response time constraints defined for the real-time system and its programs (A18-5-7).

Table 6.2: Challenged of dynamic memory usage

Ownership and smart pointers

Memory allocated dynamically requires strict control of objects or functions that are responsible for deallocating it when it is no longer needed. Such lifetime manipulation and maintenance of managing dynamically allocated memory is called Ownership. Ownership has the following features:

- if it is exclusive, then it is possible to transfer it from one scope to another.
- if it is shared, then memory deletion is typically responsibility of the last owner that releases that ownership.
- if it is temporary, then a non-owning/weak reference has to be converted to shared ownership before accessing the object.

Since C++11, management of Ownership is done by smart pointer types. Smart pointers are allocated on stack, which guarantees destructor execution and possible object deallocation (if that is the last or sole owner) at the end of the scope. Pointer-like behavior is done by overloading `operator->` and `operator*` methods.

The following standard smart pointer classes are available since C++11:

- `std::unique_ptr` wraps a pointer to an object in an exclusive way. Such `std::unique_ptr` object guarantees that only one pointer to the underlying object exists at a time, as `std::unique_ptr` is not copyable. It is movable and such an operation represents ownership transfer. When the `std::unique_ptr` instance is goes out of scope, the underlying pointer is deleted and memory is deallocated.
- `std::shared_ptr` wraps a pointer to an object in a shared way. Multiple `std::shared_ptr` are capable to point at the same object, as `std::shared_ptr` is copyable and it contains a reference counting mechanism. When the last `std::shared_ptr` instance goes out of scope (and the reference counter drops to 0), the underlying pointer is deleted and memory is deallocated.
- `std::weak_ptr` wraps a pointer to an object, but it has to be converted to a `std::shared_ptr` in order to access the referenced object. The main purpose of the `std::weak_ptr` is to break potential circular references among multiple `std::shared_ptr` objects, which would prevent reference counting from dropping to 0 and removing the underlying pointer. If only `std::weak_ptr` objects exist at a time, then conversion to `std::shared_ptr` will return an empty `std::shared_ptr`.

The main purpose of smart pointers is prevent from possible memory leaks to provide limited garbage-collection feature:

- with almost no overhead over raw pointers for `std::unique_ptr`, unless user-specified deleter is used.
- with possibility of sharing ownership among multiple `std::shared_ptr` objects.

However, this solution bases purely on the reference-counting and smart pointers destructors calling and it does not involve independent process that periodically cleans up memory that is no longer referenced.

Usage of smart pointers makes Ownership matters unambiguous and self-documenting. It also facilitates memory management issues and eliminates multiple error types that can be made by developers.

There are also other types of memory managing objects that follow RAII design pattern, e.g. std::string and std::vector.

Rule A18-5-1 (required, implementation, automated)
Functions malloc, calloc, realloc and free shall not be used.

Rationale

C-style allocation/deallocation using malloc/calloc/realloc/free functions is not type safe and does not invoke class's constructors and destructors.

Note that invoking free function on a pointer allocated with new, as well as invoking delete on a pointer allocated with malloc/realloc/calloc function, result in undefined behavior.

Also, note that realloc function should only be used on memory allocated via malloc or calloc functions.

Exception

This rule does not apply to dynamic memory allocation/deallocation performed in user-defined overloads of new and delete operators or malloc and free functions custom implementations.

Example

```
1  // $Id: A18-5-1.cpp 289436 2017-10-04 10:45:23Z michal.szczepankiewicz $
2  #include <cstdint>
3  #include <cstdlib>
4  void F1() noexcept(false)
5  {
6      // Non-compliant
7      std::int32_t* p1 = static_cast<std::int32_t*>(malloc(sizeof(std::int32_t)));
8      *p1 = 0;
9
10     // Compliant
11     std::int32_t* p2 = new std::int32_t(0);
12
13     // Compliant
14     delete p2;
15
16     // Non-compliant
17     free(p1);
18
19     // Non-compliant
20     std::int32_t* array1 =
21         static_cast<std::int32_t*>(calloc(10, sizeof(std::int32_t)));
```

```
22
23      // Non-compliant
24      std::int32_t* array2 =
25          static_cast<std::int32_t*>(realloc(array1, 10 * sizeof(std::int32_t)));
26
27      // Compliant
28      std::int32_t* array3 = new std::int32_t[10];
29
30      // Compliant
31      delete[] array3;
32
33      // Non-compliant
34      free(array2);
35
36      // Non-compliant
37      free(array1);
38  }
39  void F2() noexcept(false)
40  {
41      // Non-compliant
42      std::int32_t* p1 = static_cast<std::int32_t*>(malloc(sizeof(std::int32_t)));
43      // Non-compliant - undefined behavior
44      delete p1;
45
46      std::int32_t* p2 = new std::int32_t(0);  // Compliant
47      free(p2);  // Non-compliant - undefined behavior
48  }
49  void operator delete(void* ptr) noexcept
50  {
51      std::free(ptr);  // Compliant by exception
52  }
```

See also

- HIC++ v4.0 [9]: 5.3.2 Allocate memory using new and release it using delete.
- C++ Core Guidelines [11]: R.10: Avoid malloc() and free().

Rule A18-5-2 (required, implementation, partially automated)
Non-placement new or delete expressions shall not be used.

Rationale

If a resource returned by a non-placement new expression is assigned to a raw pointer, then a developer's mistake, an exception or a return may lead to memory leak.

It is highly recommended to follow RAII design pattern or use manager objects that manage the lifetime of variables with dynamic storage duration, e.g.:

- std::unique_ptr along with std::make_unique

- std::shared_ptr along with std::make_shared
- std::string
- std::vector

Note: Functions that do not extend lifetime shall not take parameters as smart pointers, see A8-4-11.

Exception

If the result of explicit resource allocation using a new expression is immediately passed to a manager object or an RAII class which does not provide a safe alternative for memory allocation, then it is not a violation of this rule.

This rule does not apply to dynamic memory allocation/deallocation performed in user-defined RAII classes and managers.

Placement new expression is allowed, see A18-5-10.

Example

```
// $Id: A18-5-2.cpp 316977 2018-04-20 12:37:31Z christof.meerwald $
#include <cstdint>
#include <memory>
#include <vector>
std::int32_t Fn1()
{
    std::int32_t errorCode{0};

    std::int32_t* ptr =
        new std::int32_t{0};  // Non-compliant - new expression
    // ...
    if (errorCode != 0)
    {
        throw std::runtime_error{"Error"};  // Memory leak could occur here
    }
    // ...

    if (errorCode != 0)
    {
        return 1;  // Memory leak could occur here
    }

    // ...
    delete ptr;  // Non-compilant - delete expression

    return errorCode;
}
std::int32_t Fn2()
{
    std::int32_t errorCode{0};

    std::unique_ptr<std::int32_t> ptr1 = std::make_unique<std::int32_t>(
```

```
33          0);  // Compliant - alternative for 'new std::int32_t(0)'
34
35      std::unique_ptr<std::int32_t> ptr2(new std::int32_t{
36          0});  // Non-compliant - unique_ptr provides make_unique
37                // function which shall be used instead of explicit
38                // new expression
39
40      std::shared_ptr<std::int32_t> ptr3 =
41          std::make_shared<std::int32_t>(0);  // Compliant
42
43      std::vector<std::int32_t> array;  // Compliant
44                                        // alternative for dynamic array
45
46      if (errorCode != 0)
47      {
48          throw std::runtime_error{"Error"};  // No memory leaks
49      }
50      // ...
51      if (errorCode != 0)
52      {
53          return 1;  // No memory leaks
54      }
55      // ...
56      return errorCode;  // No memory leaks
57  }
58  template <typename T>
59  class ObjectManager
60  {
61    public:
62      explicit ObjectManager(T* obj) : object{obj} {}
63      ~ObjectManager() { delete object; } // Compliant by exception
64      // Implementation
65
66    private:
67      T* object;
68  };
69  std::int32_t Fn3()
70  {
71      std::int32_t errorCode{0};
72
73      ObjectManager<std::int32_t> manager{
74          new std::int32_t{0}};  // Compliant by exception
75      if (errorCode != 0)
76      {
77          throw std::runtime_error{"Error"};  // No memory leak
78      }
79      // ...
80      if (errorCode != 0)
81      {
82          return 1;  // No memory leak
83      }
```

```
84      // ...
85      return errorCode;  // No memory leak
86  }
```

See also

- C++ Core Guidelines [11]: R.11: Avoid calling new and delete explicitly.
- C++ Core Guidelines [11]: R.12: Immediately give the result of an explicit resource allocation to a manager object.
- C++ Core Guidelines [11]: ES.60: Avoid new and delete outside resource management functions.

Rule A18-5-3 (required, implementation, automated)
The form of the delete expression shall match the form of the new expression used to allocate the memory.

Rationale

Plain and array forms of new and delete expressions must not be mixed. If an array was allocated using a new expression, then an array delete expression must be used to deallocate it and vice versa.

Example

```
1   // $Id: A18-5-3.cpp 316977 2018-04-20 12:37:31Z christof.meerwald $
2   #include <cstdint>
3
4   void Fn1()
5   {
6       std::int32_t* array =
7           new std::int32_t[10];  // new expression used to allocate an
8                                  // array object
9       // ...
10      delete array;  // Non-compliant - array delete expression supposed
11                     // to be used
12  }
13  void Fn2()
14  {
15      std::int32_t* object = new std::int32_t{0};  // new operator used to
16                                                   // allocate the memory for an
17                                                   // integer type
18      // ...
19      delete[] object;  // Non-compliant - non-array delete expression supposed
20                        // to be used
21  }
22  void Fn3()
23  {
24      std::int32_t* object = new std::int32_t{0};
```

```
25      std::int32_t* array = new std::int32_t[10];
26      // ...
27      delete[] array;  // Compliant
28      delete object;   // Compliant
29  }
```

See also

- HIC++ v4.0 [9]: 5.3.3 Ensure that the form of delete matches the form of new used to allocate the memory.

Rule A18-5-4 (required, implementation, automated)
If a project has sized or unsized version of operator "delete" globally defined, then both sized and unsized versions shall be defined.

Rationale

Since C++14 Language Standard it is allowed to overload both sized and unsized versions of the "delete" operator. Sized version provides more efficient way of memory deallocation than the unsized one, especially when the allocator allocates in size categories instead of storing the size nearby the object.

Example

```
1  //% $Id: A18-5-4.cpp 289415 2017-10-04 09:10:20Z piotr.serwa $
2  #include <cstdlib>
3  void operator delete(
4      void* ptr) noexcept  // Compliant - sized version is defined
5  {
6      std::free(ptr);
7  }
8  void operator delete(
9      void* ptr,
10     std::size_t size) noexcept  // Compliant - unsized version is defined
11 {
12     std::free(ptr);
13 }
```

See also

- none

Rule A18-5-5 (required, toolchain, partially automated)
Memory management functions shall ensure the following: (a) deterministic behavior resulting with the existence of worst-case execution time, (b) avoiding memory fragmentation, (c) avoid running out of memory, (d)

avoiding mismatched allocations or deallocations, (e) no dependence on non-deterministic calls to kernel.

Rationale

Memory management errors occur commonly and they can affect application stability and correctness. The main problems of dynamic memory management are as following:

- Non deterministic worst-case execution time of allocation and deallocation
- Invalid memory access
- Mismatched allocations and deallocations
- Memory fragmentation
- Running out of memory

Custom memory management functions (custom allocators) need to address all of this problems for the project and all libraries used in the project.

To ensure the worst-case execution time, the memory management functions need to be executed without context switch and without syscalls.

To prevent running out of memory, an executable is supposed to define its maximal memory needs, which are pre-allocated for this executable during its startup.

Memory management functions include operators new and delete, as well as low-level functions malloc and free. Nevertheless code written in C++ language uses new and delete operators, and direct use of malloc and free operations do not occur, some libraries, e.g. exception handling mechanism routines of libgcc uses malloc and free functions directly and omits new and delete operators usage. Custom memory management functionality needs to provide custom implementation of C++ new and delete operators, as well as implementation of malloc and free operations to hide incorrect dynamic memory allocation/deallocation in linked libraries.

Note: If custom memory management requires to use custom std::new_handler, its implementation shall perform one of the following:

- make more storage available for allocation and then return
- throw an exception of type bad_alloc or a class derived from bad_alloc
- terminate execution of the program without returning to the caller

Example

```
1 //% $Id: A18-5-5.cpp 289815 2017-10-06 11:19:11Z michal.szczepankiewicz $
2
3 #define __GNU_SOURCE
4 #include <dlfcn.h>
5 #include <cstddef>
6
```

```
7  void* MallocBad(size_t size)  // Non-compliant, malloc from libc does not
8                                // guarantee deterministic execution time
9  {
10     void* (*libcMalloc)(size_t) = (void* (*)(size_t))dlsym(RTLD_NEXT, "malloc");
11     return libcMalloc(size);
12 }
13
14 void FreeBad(void* ptr)  // Non-compliant, malloc from libc does not guarantee
15                          // deterministic execution time
16 {
17     void (*libcFree)(void*) = (void (*)(void*))dlsym(RTLD_NEXT, "free");
18     libcFree(ptr);
19 }
20
21 void* MallocGood(size_t size)  // Compliant - custom malloc implementation that
22                                // will guarantee deterministic execution time
23 {
24     // Custom implementation that provides deterministic worst-case execution
25     // time
26 }
27
28 void FreeGood(void* ptr)  // Compliant - custom malloc implementation that will
29                           // guarantee deterministic execution time
30 {
31     // Custom implementation that provides deterministic worst-case execution
32     // time
33 }
```

See also

- none

Rule A18-5-6 (required, verification / toolchain, non-automated)
An analysis shall be performed to analyze the failure modes of dynamic memory management. In particular, the following failure modes shall be analyzed: (a) non-deterministic behavior resulting with nonexistence of worst-case execution time, (b) memory fragmentation, (c) running out of memory, (d) mismatched allocations and deallocations, (e) dependence on non-deterministic calls to kernel.

Rationale

The worst-case execution time and behavior of memory management functions are specific to each implementation. In order to use dynamic memory in the project, an analysis needs to be done to determine possible errors and worst-case execution time of allocation and deallocation functions.

Note that standard C++ implementation violates some of this requirements. However, listed problems can be addressed by implementing or using a custom memory allocator.

See also

- none

Rule A18-5-7 (required, implementation, non-automated)
If non-realtime implementation of dynamic memory management functions is used in the project, then memory shall only be allocated and deallocated during non-realtime program phases.

Rationale

If worst-case execution time of memory management functions can not be determined, then dynamic memory usage is prohibited during realtime program phase, but it can be used e.g. during initialization or non-realtime state transitions.

See: Real-time.

Example

```
1  //% $Id: A18-5-7.cpp 289436 2017-10-04 10:45:23Z michal.szczepankiewicz $
2  #include <cstdint>
3  #include <memory>
4  #include <vector>
5  std::int8_t AppMainLoop() noexcept
6  {
7      std::int8_t retCode = 0;
8      std::int32_t* arr[10];
9      while (true)
10     {
11         for (std::int8_t i = 0; i < 10; ++i)
12         {
13             arr[i] = new std::int32_t{
14                 i};  // Non-compliant - allocation in a phase that
15                      // requires real-time
16         }
17         // Implementation
18         for (auto& i : arr)
19         {
20             delete i;  // Non-compliant - deallocation in a phase that requires
21                        // real-time
22         }
23     }
24     return retCode;
25 }
26 static std::int32_t* object =
27     new std::int32_t{0};  // Compliant- allocating in start-up phase
28
29 int main(int, char**)
30 {
31     std::unique_ptr<std::int32_t> ptr =
32         std::make_unique<std::int32_t>(0);  // Compliant
```

```
33      std::vector<std::int32_t> vec;          // Compliant
34      vec.reserve(10);                        // Compliant
35
36      std::int8_t code = AppMainLoop();
37      return code;
38  }
```

See also

- none

Rule A18-5-8 (required, implementation, partially automated)
Objects that do not outlive a function shall have automatic storage duration.

Rationale

Creating objects with automatic storage duration implies that there is no additional allocation and deallocation cost, which would occur when using dynamic storage.

Note: This rule applies only to objects created in a function scope, it does not forbid the object to internally allocate additional memory on heap.

Exception

Objects causing high memory utilization may be allocated on heap using memory managing objects.

Example

```
1  //% $Id: A18-5-8.cpp 311792 2018-03-15 04:15:08Z christof.meerwald $
2  #include <cstdint>
3  #include <memory>
4  #include <vector>
5
6  class StackBitmap
7  {
8    public:
9      constexpr static size_t maxSize = 65535;
10     using BitmapRawType = std::array<uint8_t, maxSize>;
11     StackBitmap(const std::string& path, uint32_t bitmapSize)
12     {
13         // read bitmapSize bytes from the file path
14     }
15
16     const BitmapRawType& GetBitmap() const noexcept { return bmp; }
17
18   private:
19     BitmapRawType bmp;
20 };
21
```

```
22 void AddWidgetToLayout(int32_t row, int32_t col)
23 {
24     auto idx = std::make_pair(row, col);  // Compliant
25     auto spIdx = std::make_shared<std::pair<int32_t, int32_t>>(
26         row, col);  // Non-compliant
27     // addWidget to index idx
28 }
29
30 uint8_t CalcAverageBitmapColor(const std::string& path, uint32_t bitmapSize)
31 {
32     std::vector<uint8_t> bmp1(bitmapSize);  // Compliant
33     // read bitmap from path
34     StackBitmap bmp2(path, bitmapSize);  // Non-compliant
35     bmp2.GetBitmap();
36 }
37
38 int main(int, char**)
39 {
40     AddWidgetToLayout(5, 8);
41     CalcAverageBitmapColor("path/to/bitmap.bmp", 32000);
42 }
```

See also

- C++ Core Guidelines [11]: R.5: Prefer scoped objects, don't heap-allocate unnecessarily.

Rule A18-5-9 (required, implementation, automated)
Custom implementations of dynamic memory allocation and deallocation functions shall meet the semantic requirements specified in the corresponding "Required behaviour" clause from the C++ Standard.

Rationale

It is possible to provide custom implementations of global dynamic memory allocation/deallocation functions. Requirements for custom implementations for each function declaration are specified in the C++ Standard in the section "Required behaviour". If the provided function do not implement the required semantics, it can lead to undefined behaviour.

Example

```
1 //% $Id: A18-5-9.cpp 305629 2018-01-29 13:29:25Z piotr.serwa $
2 #include <new>
3
4 void* operator new(std::size_t count, const std::nothrow_t& tag)
5 {
6     extern void* custom_alloc(std::size_t); // Implemented elsewhere; may return
      nullptr
7     if (void *ret = custom_alloc(count))
```

```
8       {
9           return ret;
10      }
11      throw std::bad_alloc(); //non-compliant, this version of new method shall not
         throw exceptions
12  }
```

See also

- SEI CERT C++ Coding Standard [10]: MEM55-CPP: Honor replacement dynamic storage management requirements

Rule A18-5-10 (required, implementation, automated)
Placement new shall be used only with properly aligned pointers to sufficient storage capacity.

Rationale

Placement new can be useful for cases in which allocation is required separately from type initialization, e.g. memory allocators, generic containers. Correct usage of placement new requires passing a pointer that:

- is suitably aligned
- provides sufficient storage memory

Violating above constrains will result in an object constructed at a misaligned location or memory initialization outside of the allocated bounds, which leads to undefined behaviour. An initial memory pointer used for placement new shall not be used after the call.

Example

```
1   //% $Id: A18-5-10.cpp 305629 2018-01-29 13:29:25Z piotr.serwa $
2
3   #include <new>
4   #include <cstdint>
5
6   void Foo()
7   {
8       uint8_t c;
9       uint64_t* ptr = ::new (&c) uint64_t;
10      //non-compliant, insufficient storage
11  }
12
13  void Bar()
14  {
15      uint8_t c; // Used elsewhere in the function
16      uint8_t buf[sizeof(uint64_t)];
17      uint64_t* ptr = ::new (buf) uint64_t;
```

```
18      //non-compliant, storage not properly aligned
19  }
```

See also

- SEI CERT C++ Coding Standard [10]: MEM54-CPP: Provide placement new with properly aligned pointers to sufficient storage capacity

Rule A18-5-11 (required, implementation, automated)
"operator new" and "operator delete" shall be defined together.

Rationale

Providing a custom allocation function (`operator new`) for a class or program implies the use of a custom memory management scheme different to the default one. It is therefore unlikely that memory allocated using a custom allocation function can be deallocated by the default deallocation function (`operator delete`).

Example

```
1  //% $Id: A18-5-11.cpp 316977 2018-04-20 12:37:31Z christof.meerwald $
2  #include <cstdlib>
3
4  class A {
5  public:
6      static void * operator new(std::size_t s); // Compliant: operator new
7      static void operator delete(void *ptr);    // defined together with
8                                                 // operator delete
9  };
10
11 class B {
12 public:
13     static void * operator new(std::size_t s);     // Non-compliant: operator
14     static void * operator new [](std::size_t s); // new defined without
15                                                    // corresponding operator
16                                                    // delete
17 };
```

See also

- HIC++ v4.0 [9]: 12.3.1: Correctly declare overloads for operator new and delete

6.18.9 Other runtime support

Rule M18-7-1 (required, implementation, automated)
The signal handling facilities of <csignal> shall not be used.

See MISRA C++ 2008 [7]

Rule A18-9-1 (required, implementation, automated)
The std::bind shall not be used.

Rationale

Using the std::bind function makes the function call less readable and may lead to the developer confusing one function parameter with another. Also, compilers are less likely to inline the functions that are created using std::bind.

It is recommended to use lambda expressions instead.

Example

```
// $Id: A18-9-1.cpp 289436 2017-10-04 10:45:23Z michal.szczepankiewicz $
#include <cstdint>
#include <functional>
class A
{
    // Implementation
};
void Fn(A const& a, double y) noexcept
{
    // Implementation
}
void F1() noexcept
{
    double y = 0.0;
    auto function = std::bind(&Fn, std::placeholders::_1, y);  // Non-compliant
    // ...
    A const a{};
    function(a);
}
void F2() noexcept
{
    auto lambda = [](A const& a) -> void {
        double y = 0.0;
        Fn(a, y);
    };  // Compliant
    // ...
    A const a{};
    lambda(a);
}
```

See also

- Effective Modern C++ [13]: Item 34: Prefer lambdas to std::bind

Rule A18-9-2 (required, implementation, automated)
Forwarding values to other functions shall be done via: (1) std::move if the value is an rvalue reference, (2) std::forward if the value is forwarding reference.

Rationale

The std::move function unconditionally casts an rvalue reference to rvalue, while the std::forward function does the same if and only if the argument was initialized with an rvalue. Both functions should be used as follows:

- std::move should be used for forwarding rvalue references to other functions, as rvalue reference always bounds to rvalue
- std::forward should be used for forwarding forwarding references to other functions, as forwarding reference might be bound to lvalue or rvalue

Note that parameter of type “auto&&” is also considered as a forwarding reference for the purpose of this rule.

Example

```
1  // $Id: A18-9-2.cpp 289436 2017-10-04 10:45:23Z michal.szczepankiewicz $
2  #include <cstdint>
3  #include <string>
4  #include <utility>
5  class A
6  {
7    public:
8      explicit A(std::string&& s)
9               : str(std::move(s))  // Compliant - forwarding rvalue reference
10     {
11     }
12
13   private:
14     std::string str;
15 };
16 class B
17 {
18 };
19 void Fn1(const B& lval)
20 {
21 }
22 void Fn1(B&& rval)
23 {
24 }
25 template <typename T>
26 void Fn2(T&& param)
27 {
28     Fn1(std::forward<T>(param));  // Compliant - forwarding forwarding reference
29 }
30 template <typename T>
```

```
void Fn3(T&& param)
{
    Fn1(std::move(param));  // Non-compliant - forwarding forwarding reference
                            // via std::move
}
void Fn4() noexcept
{
    B b1;
    B& b2 = b1;
    Fn2(b2);              // fn1(const B&) is called
    Fn2(std::move(b1));   // fn1(B&&) is called
    Fn3(b2);              // fn1(B&&) is called
    Fn3(std::move(b1));   // fn1(B&&) is called
}
```

See also

- HIC++ v4.0 [9]:17.3.2 Use std::forward to forward universal references
- Effective Modern C++ [13]: Item 25. Use std::move on rvalue references, std::forward on universal references.

Rule A18-9-3 (required, implementation, automated)
The std::move shall not be used on objects declared const or const&.

Rationale

If an object is declared const or const&, then it will actually never be moved using the std::move.

Example

```
// $Id: A18-9-3.cpp 289436 2017-10-04 10:45:23Z michal.szczepankiewicz $
#include <utility>
class A
{
    // Implementation
};
void F1()
{
    const A a1{};
    A a2 = a1;              // Compliant - copy constructor is called
    A a3 = std::move(a1);   // Non-compliant - copy constructor is called
                            // implicitly instead of move constructor
}
```

See also

- HIC++ v4.0 [9]: 17.3.1 Do not use std::move on objects declared with const or const& type.

Rule A18-9-4 (required, implementation, automated)
An argument to std::forward shall not be subsequently used.

Rationale

Depending on the value category of parameters used in the call, std::forward may result in a move of the parameter. When the value is an lvalue, modifications to the parameter will affect the argument of the caller. If the value is an rvalue, the value may be in indeterminate state after the call to std::forward.

Example

```
1  // $Id: A18-9-4.cpp 289436 2017-10-04 10:45:23Z michal.szczepankiewicz $
2  #include <cstdint>
3  #include <iostream>
4  #include <utility>
5  template <typename T1, typename T2>
6  void F1(T1 const& t1, T2& t2){
7      // ...
8  };
9  template <typename T1, typename T2>
10 void F2(T1&& t1, T2&& t2)
11 {
12     f1(std::forward<T1>(t1), std::forward<T2>(t2));
13     ++t2;  // Non-compliant
14 };
```

See also

- HIC++ v4.0 [9]: 17.3.3 Do not subsequently use the argument to std::forward.

6.19 Diagnostics library - partial

6.19.4 Error numbers

Rule M19-3-1 (required, implementation, automated)
The error indicator errno shall not be used.

See MISRA C++ 2008 [7]

6.20 General utilities library - partial

6.20.8 Smart pointers

Rule A20-8-1 (required, implementation, automated)
An already-owned pointer value shall not be stored in an unrelated smart pointer.

Rationale

Smart pointers (e.g. `std::shared_ptr`) that allow to manage the same underlying pointer value using multiple smart pointer objects, shall be created in a way that creates a relationship between two smart pointer objects (e.g. via copy assignment). Unrelated smart pointer objects with a pointer value that is owned by another smart pointer object shall not be created.

Example

```
1  // $Id: A20-8-1.cpp 305588 2018-01-29 11:07:35Z michal.szczepankiewicz $
2
3  #include <memory>
4
5  void Foo()
6  {
7      uint32_t *i = new uint32_t{5};
8      std::shared_ptr<uint32_t> p1(i);
9      std::shared_ptr<uint32_t> p2(i); // non-compliant
10 }
11
12 void Bar()
13 {
14     std::shared_ptr<uint32_t> p1 = std::make_shared<uint32_t>(5);
15     std::shared_ptr<uint32_t> p2(p1); //compliant
16 }
```

See also

- SEI CERT C++ Coding Standard [10]: MEM56-CPP: Do not store an already-owned pointer value in an unrelated smart pointer

Rule A20-8-2 (required, implementation, automated)
A std::unique_ptr shall be used to represent exclusive ownership.

Rationale

`std::unique_ptr` is a smart pointer that owns and manages another object and removes it when it goes out of scope. It has almost no overhead over a raw pointer and clearly states developers intentions and ownership status of the object.

Note: Further usage of the instance of `std::unique_ptr` in another scope requires transferring ownership using move semantics.

Example

```
// $Id: A20-8-2.cpp 308981 2018-02-26 08:11:52Z michal.szczepankiewicz $

#include <thread>
#include <memory>

struct A
{
    A(std::uint8_t xx, std::uint8_t yy) : x(xx), y(yy) {}
    std::uint8_t x;
    std::uint8_t y;
};

//consumes object obj or just uses it
void Foo(A* obj) { }
void Bar(std::unique_ptr<A> obj) { }

int main(void)
{
    A* a = new A(3,5); //non-compliant with A18-5-2
    std::unique_ptr<A> spA = std::make_unique<A>(3,5);

    //non-compliant, not clear if function assumes
    //ownership of the object
    std::thread th1{&Foo, a};
    std::thread th2{&Foo, a};
    //compliant, it is clear that function Bar
    //assumes ownership
    std::thread th3{&Bar, std::move(spA)};

    th1.join();
    th2.join();
    th3.join();
    return 0;
}
```

See also

- JSF December 2005 [8]: AV Rule 112: Function return values should not obscure resource ownership.
- C++ Core Guidelines [11]: F.26: Use a unique_ptr<T> to transfer ownership where a pointer is needed
- C++ Core Guidelines [11]: R.20: Use unique_ptr or shared_ptr to represent ownership

Rule A20-8-3 (required, implementation, automated)
A std::shared_ptr shall be used to represent shared ownership.

Rationale

`std::shared_ptr` allows to retain shared ownership by keeping multiple `std::shared_ptr` instances pointing at the same object. The object is removed when the last `std::shared_ptr` instance goes out of scope. Although reference counting mechanism included brings some overhead over a raw pointer, it clearly states ownership status of the object and effectively prevents from possible memory leaks.

Example

```
1  // $Id: A20-8-3.cpp 308507 2018-02-21 13:23:57Z michal.szczepankiewicz $
2
3  #include <memory>
4  #include <cstdint>
5  #include <thread>
6
7  struct A
8  {
9      A(std::uint8_t xx, std::uint8_t yy) : x(xx), y(yy) {}
10     std::uint8_t x;
11     std::uint8_t y;
12 };
13
14 void Foo(A* obj) { }
15 void Bar(A* obj) { }
16
17 void Foo2(std::shared_ptr<A> obj) { }
18 void Bar2(std::shared_ptr<A> obj) { }
19
20 int main(void)
21 {
22     A* a = new A(3,5); //non-compliant with A18-5-2
23     std::shared_ptr<A> spA = std::make_shared<A>(3,5);
24
25     //non-compliant, not clear who is responsible
26     //for deleting object a
27     std::thread th1{&Foo, a};
28     std::thread th2{&Bar, a};
29
30     //compliant, object spA gets deleted
31     //when last shared_ptr gets destructed
32     std::thread th3{&Foo2, spA};
33     std::thread th4{&Bar2, spA};
34
35     th1.join();
36     th2.join();
37     th3.join();
38     th4.join();
39
40     return 0;
41 }
```

See also

- JSF December 2005 [8]: AV Rule 112: Function return values should not obscure resource ownership.
- C++ Core Guidelines [11]: F.27: Use a shared_ptr<T> to share ownership
- C++ Core Guidelines [11]: R.20: Use unique_ptr or shared_ptr to represent ownership

Rule A20-8-4 (required, implementation, automated)
A std::unique_ptr shall be used over std::shared_ptr if ownership sharing is not required.

Rationale

`std::unique_ptr` is more predictable in terms of its destruction, as it happens at the end of the scope unless ownership transfer occurred. It also has lower overhead than a `std::shared_ptr`, as it does not keep internal reference counting.

Example

```
1  // $Id: A20-8-4.cpp 308507 2018-02-21 13:23:57Z michal.szczepankiewicz $
2
3  #include <memory>
4  #include <cstdint>
5  #include <thread>
6
7  struct A
8  {
9      A(std::uint8_t xx, std::uint8_t yy) : x(xx), y(yy) {}
10     std::uint8_t x;
11     std::uint8_t y;
12 };
13
14 void Func()
15 {
16     auto spA = std::make_shared<A>(3,5);
17     //non-compliant, shared_ptr used only locally
18     //without copying it
19 }
20
21 void Foo(std::unique_ptr<A> obj) { }
22 void Bar(std::shared_ptr<A> obj) { }
23
24 int main(void)
25 {
26     std::shared_ptr<A> spA = std::make_shared<A>(3,5);
27     std::unique_ptr<A> upA = std::make_unique<A>(4,6);
28
29     //compliant, object accesses in parallel
```

```
30      std::thread th1{&Bar, spA};
31      std::thread th2{&Bar, spA};
32      std::thread th3{&Bar, spA};
33
34      //compliant, object accesses only by 1 thread
35      std::thread th4{&Foo, std::move(upA)};
36
37      th1.join();
38      th2.join();
39      th3.join();
40      th4.join();
41
42      return 0;
43  }
```

See also

- C++ Core Guidelines [11]: R.21: Prefer unique_ptr over shared_ptr unless you need to share ownership

Rule A20-8-5 (required, implementation, automated)
std::make_unique shall be used to construct objects owned by std::unique_ptr.

Rationale

Using `std::make_unique` to create instances of `std::unique_ptr<T>` provides object allocation without explicit call of `new` function, see A18-5-2. It also ensures exception safety in complex expressions and prevents from memory leaks caused by unspecified-evaluation order-expressions.

Exception

It is allowed to use explicit `new` function call to create an instance of `std::unique_ptr<T>`, if it requires a custom deleter.

Example

```
1  // $Id: A20-8-5.cpp 308507 2018-02-21 13:23:57Z michal.szczepankiewicz $
2  #include <memory>
3  #include <cstdint>
4  #include <functional>
5
6  struct A
7  {
8      A() { throw std::runtime_error("example"); }
9      A(std::uint8_t xx, std::uint8_t yy) : x(xx), y(yy) {}
10     std::uint8_t x;
11     std::uint8_t y;
12 };
```

```
13
14 void Foo(std::unique_ptr<A> a, std::unique_ptr<A> b) { }
15
16 int main(void)
17 {
18     //compliant
19     std::unique_ptr<A> upA = std::make_unique<A>(4,6);
20     //non-compliant
21     std::unique_ptr<A> upA2 = std::unique_ptr<A>(new A(5,7));
22
23     //non-compliant, potential memory leak, as A class constructor throws
24     Foo(std::unique_ptr<A>(new A()), std::unique_ptr<A>(new A()));
25     //non-compliant, potential memory leak, as A class constructor throws
26     Foo(std::make_unique<A>(4,6), std::unique_ptr<A>(new A()));
27     //compliant, no memory leaks
28     Foo(std::make_unique<A>(4,6), std::make_unique<A>(4,6));
29
30     //compliant by exception
31     std::unique_ptr<A, std::function<void(A*)>> ptr(new A(4,5), [](A* b) { delete
       b; } );
32
33     return 0;
34 }
```

See also

- C++ Core Guidelines [11]: R.23: Use make_unique() to make unique_ptrs
- C++ Core Guidelines [11]: C.150: Use make_unique() to construct objects owned by unique_ptrs

Rule A20-8-6 (required, implementation, automated)
std::make_shared shall be used to construct objects owned by std::shared_ptr.

Rationale

`std::shared_ptr` manages two entities: a control block (for meta data such as reference counter or type-erased deleter) and an allocated object. Using `std::make_shared` typically performs a single heap allocation (as it is recommended by the Standard) for both control block and allocated object. `std::make_shared` function also provides object allocation without explicit call of `new` function, see A18-5-2. It also ensures exception safety and prevents from memory leaks caused by unspecified-evaluation-order expressions.

Exception

It is allowed to use explicit `new` function call to create an instance of `std::shared_ptr`, if it requires a custom deleter. It is also allowed to construct objects owned by `std::shared_ptr` using `std::allocate_shared`.

Example

```
// $Id: A20-8-6.cpp 308507 2018-02-21 13:23:57Z michal.szczepankiewicz $
#include <memory>
#include <cstdint>
#include <functional>

struct A
{
    A() { throw std::runtime_error("example"); }
    A(std::uint8_t xx, std::uint8_t yy) : x(xx), y(yy) {}
    std::uint8_t x;
    std::uint8_t y;
};

void Foo(std::shared_ptr<A> a, std::shared_ptr<A> b) { }

int main(void)
{
    //compliant
    std::shared_ptr<A> upA = std::make_shared<A>(4,6);
    //non-compliant
    std::shared_ptr<A> upA2 = std::shared_ptr<A>(new A(5,7));

    //non-compliant, potential memory leak, as A class constructor throws
    Foo(std::shared_ptr<A>(new A()), std::shared_ptr<A>(new A()));
    //non-compliant, potential memory leak, as A class constructor throws
    Foo(std::make_shared<A>(4,6), std::shared_ptr<A>(new A()));
    //compliant, no memory leaks
    Foo(std::make_shared<A>(4,6), std::make_shared<A>(4,6));

    //compliant by exception
    std::shared_ptr<A> ptr(new A(4,5), [](A* b) { delete b; } );

    return 0;
}
```

See also

- C++ Core Guidelines [11]: R.22: Use make_shared() to make shared_ptrs.
- C++ Core Guidelines [11]: C.151: Use make_shared() to construct objects owned by shared_ptrs

Rule A20-8-7 (required, implementation, non-automated)
A std::weak_ptr shall be used to represent temporary shared ownership.

Rationale

A cyclic structure of `std::shared_ptr` results in reference counting mechanism never dropping to zero, which prevents from pointed object deallocation. Breaking such

cycles is done using `std::weak_ptr` which must be converted to `std::shared_ptr` in order to access the referenced object.

Example

```
// $Id: A20-8-7.cpp 308795 2018-02-23 09:27:03Z michal.szczepankiewicz $

#include <memory>

template <template <typename> class T, typename U>
struct Base
{
    T<U> sp;
};

template <typename T>
using Shared = Base<std::shared_ptr, T>;

template <typename T>
using Weak = Base<std::weak_ptr, T>;

struct SBarSFoo;
struct SFooSBar : public Shared<SBarSFoo> {};
struct SBarSFoo : public Shared<SFooSBar> {};

struct A : public Shared<A> {};

struct WBarSFoo;
struct SFooWBar : public Shared<WBarSFoo> {};
struct WBarSFoo : public Weak<SFooWBar> {};

int main()
{
    std::shared_ptr<SFooSBar> f = std::make_shared<SFooSBar>();
    std::shared_ptr<SBarSFoo> b = std::make_shared<SBarSFoo>();
    f->sp = b;
    b->sp = f;
    //non-compliant, both f and b have ref_count() == 2
    //destructors of f and b reduce ref_count() to 1,
    //destructors of underlying objects are never called,
    //so destructors of shared_ptrs sp are not called
    //and memory is leaked

    std::shared_ptr<A> a = std::make_shared<A>();
    a->sp = a;
    //non-compliant, object 'a' destructor does not call
    //underlying memory destructor

    std::shared_ptr<SFooWBar> f2 = std::make_shared<SFooWBar>();
    std::shared_ptr<WBarSFoo> b2 = std::make_shared<WBarSFoo>();
    f2->sp = b2;
    b2->sp = f2;
```

```
48      //compliant, b2->sp holds weak_ptr to f2, so f2 destructor
49      //is able to properly destroy underlying object
50
51      return 0;
52  }
```

See also

- C++ Core Guidelines [11]: R.24: Use std::weak_ptr to break cycles of shared_ptrs

6.21 Strings library

6.21.8 Null-terminated sequence utilities

Rule A21-8-1 (required, implementation, automated)
Arguments to character-handling functions shall be representable as an unsigned char.

Rationale

This rule applies to the character handling functions in `<cctype>`. They all take an `int` parameter as input and specify that its value either shall be `EOF` or otherwise shall be representable as an `unsigned char`. On platforms where `char` is signed, it can have negative values that are not representable as an `unsigned char`, so that passing a `char` to such a function can result in undefined behavior.

Thus, this rule mandates that all character arguments passed to such functions shall be explicitly cast to `unsigned char`.

Note: Of all the functions in `<cctype>`, `isdigit` and `isxdigit` are the only ones whose behavior does not depend on the currently installed locale. See A18-0-3 in section 6.18.0 for a rule concerning the `setlocale` function.

Example

```
1  // $Id: A21-8-1.cpp 312606 2018-03-21 09:52:14Z jan.babst $
2  #include <algorithm>
3  #include <cctype>
4  #include <string>
5
6  void RemoveDigits_Bad(std::string& s)
7  {
8      s.erase(std::remove_if(s.begin(),
9                             s.end(),
10                            [](char c) {
11                                return std::isdigit(c);  // non-compliant
12                            }),
```

```
            s.cend());
}

void RemoveDigits_Good(std::string& s)
{
    s.erase(std::remove_if(s.begin(),
                           s.end(),
                           [](char c) {
                               return std::isdigit(
                                   static_cast<unsigned char>(c));  // compliant
                           }),
            s.cend());
}
```

See also

- SEI CERT C++ Coding Standard [10]: STR37-C: Arguments to character-handling functions must be representable as an unsigned char.
- cppreference.com [16]: Standard library header <cctype>.
- Rule A18-0-3 in section 6.18.0

6.23 Containers library - partial

6.23.1 General

Rule A23-0-1 (required, implementation, automated)
An iterator shall not be implicitly converted to const_iterator.

Rationale

The Standard Template Library introduced methods for returning const iterators to containers. Making a call to these methods and immediately assigning the value they return to a const_iterator, removes implicit conversions.

Example

```
//% $Id: A23-0-1.cpp 289436 2017-10-04 10:45:23Z michal.szczepankiewicz $
#include <cstdint>
#include <vector>

void Fn1(std::vector<std::int32_t>& v) noexcept
{
    for (std::vector<std::int32_t>::const_iterator iter{v.cbegin()},
         end{v.cend()};
         iter != end;
         ++iter)  // Compliant
    {
```

```
12          // ...
13      }
14  }
15
16  void Fn2(std::vector<std::int32_t>& v) noexcept
17  {
18      for (auto iter{v.cbegin()}, end{v.cend()}; iter != end;
19           ++iter)  // Compliant
20      {
21          // ...
22      }
23  }
24
25  void Fn3(std::vector<std::int32_t>& v) noexcept
26  {
27      for (std::vector<std::int32_t>::const_iterator iter{v.begin()},
28           end{v.end()};
29           iter != end;
30           ++iter)  // Non-compliant
31      {
32          // ...
33      }
34  }
```

See also

- HIC++ v4.0 [9]: 17.4.1 Use const container calls when result is immediately converted to a const iterator.

Rule A23-0-2 (required, implementation, automated)
Elements of a container shall only be accessed via valid references, iterators, and pointers.

Rationale

Some operations on standard library containers invalidate references, iterators, and pointers to container elements which were previously stored.

The behavior of the standard library containers and their operations with respect to the invalidation of references, iterators, and pointers is well documented, e.g. in [16].

Example

```
1  // $Id: A23-0-2.cpp 309868 2018-03-02 10:47:23Z jan.babst $
2  #include <algorithm>
3  #include <cstdint>
4  #include <list>
5  #include <vector>
6
7  void f()
```

```
{
    std::vector<int32_t> v{0, 1, 2, 3, 4, 5, 6, 7};

    auto it = std::find(v.begin(), v.end(), 5);  // *it is 5

    // These calls may lead to a reallocation of the vector storage
    // and thus may invalidate the iterator it.
    v.push_back(8);
    v.push_back(9);

    *it = 42;  // Non-compliant
}

void g()
{
    std::list<int32_t> l{0, 1, 2, 3, 4, 5, 6, 7};

    auto it = std::find(l.begin(), l.end(), 5);  // *it is 5
    l.remove(7);
    l.push_back(9);
    *it = 42;  // Compliant - previous operations do not invalidate iterators
    // l is now {0, 1, 2, 3, 4, 42, 6, 9 }
}
```

See also

- SEI CERT C++ Coding Standard [10]: CTR51-CPP: Use valid references, pointers, and iterators to reference elements of a container.
- SEI CERT C++ Coding Standard [10]: STR52-CPP: Use valid references, pointers, and iterators to reference elements of a basic_string.
- cppreference.com [16]: Containers library. Iterator invalidation.

6.25 Algorithms library

6.25.1 General

Rule A25-1-1 (required, implementation, automated)
Non-static data members or captured values of predicate function objects that are state related to this object's identity shall not be copied.

Rationale

Generic algorithms available in the C++ Standard Library accept a predicate function object. The ISO/IEC 14882:2014 C++ Language Standard states that it is implementation-defined whether predicate function objects can be copied by the STL algorithms.

To prevent from unexpected results while using predicate function objects, any such object shall either:

- be passed to an STL algorithm wrapped as a `std::reference_wrapper`.
- implement a function call operator that is `const` and does not modify any data members or captured values that have a mutable specifier.

Example

```
1  //% $Id: A25-1-1.cpp 309784 2018-03-01 20:18:29Z michal.szczepankiewicz $
2
3  #include <iostream>
4  #include <vector>
5  #include <algorithm>
6  #include <functional>
7  #include <iterator>
8
9  class ThirdElemPred : public std::unary_function<int, bool>
10 {
11 public:
12     ThirdElemPred() : timesCalled(0) {}
13     bool operator()(const int &) { return (++timesCalled) == 3; }
14     //non-compliant, non-const call operator that
15     //modifies the predicate object field
16 private:
17     size_t timesCalled;
18 };
19
20 class ThirdElemPred2 : public std::unary_function<int, bool>
21 {
22 public:
23     ThirdElemPred2() : timesCalled(0) {}
24     bool operator()(const int &) const { return (++timesCalled) == 3; }
25     //non-compliant, const call operator that
26     //modifies the mutable predicate object field
27 private:
28     mutable size_t timesCalled;
29 };
30
31 class ValueFivePred: public std::unary_function<int, bool>
32 {
33 public:
34     bool operator()(const int& v) const { return v == 5; }
35     //compliant, const call operator that does not
36     //modify the predicate object state
37 };
38
39 void F1(std::vector<int> v)
40 {
41     //non-compliant, predicate object state modified
42     int timesCalled = 0;
43     //display values that are NOT to be removed
```

```
44      std::copy(v.begin(), std::remove_if(v.begin(), v.end(), [timesCalled](const
        int &) mutable { return (++timesCalled) == 3; }), std::ostream_iterator<std::
        vector<int>::value_type>(std::cout, " ") );
45      std::cout << std::endl;
46  }
47
48  void F2(std::vector<int> v)
49  {
50      //non-compliant, predicate object state modified
51      std::copy(v.begin(), std::remove_if(v.begin(), v.end(), ThirdElemPred()), std
        ::ostream_iterator<std::vector<int>::value_type>(std::cout, " ") );
52      std::cout << std::endl;
53  }
54
55  void F22(std::vector<int> v)
56  {
57      //non-compliant, predicate object state modified
58      std::copy(v.begin(), std::remove_if(v.begin(), v.end(), ThirdElemPred2()),
        std::ostream_iterator<std::vector<int>::value_type>(std::cout, " ") );
59      std::cout << std::endl;
60  }
61
62  void F3(std::vector<int> v)
63  {
64      //compliant, predicate object that has its state
65      //modified is passed as a std::reference_wrapper
66      ThirdElemPred pred;
67      std::copy(v.begin(), std::remove_if(v.begin(), v.end(), std::ref(pred)), std
        ::ostream_iterator<std::vector<int>::value_type>(std::cout, " ") );
68      std::cout << std::endl;
69  }
70
71  int main(void)
72  {
73      std::vector<int> v{0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
74
75      F1(v);
76      F2(v);
77      F22(v);
78      F3(v);
79  //output for g++-5.5, correct result only for F3
80  //F1    0 1 3 4 6 7 8 9
81  //F2    0 1 3 4 6 7 8 9
82  //F22   0 1 3 4 6 7 8 9
83  //F3    0 1 3 4 5 6 7 8 9
84      return 0;
85  }
```

See also

- SEI CERT C++ Coding Standard [10]: CTR58-CPP: Predicate function objects should not be mutable
- cppreference.com [16]: C++ concepts: Predicate

6.25.4 Sorting and related operations

Rule A25-4-1 (required, implementation, non-automated)
Ordering predicates used with associative containers and STL sorting and related algorithms shall adhere to a strict weak ordering relation.

Rationale

Ordering predicates that can be passed to associative containers or sorting STL algorithms and related operations must fulfill requirements for a strict weak ordering, e.g.:

- irreflexivity: FOR ALL `x: x < x == false`
- assymetry: FOR ALL `x, y: if x < y then !(y < x)`
- transitivity: FOR ALL `x, y, z: if x < y && y < z then x < z`

Ordering predicates not adhering to these requirements will result in these algorithms not working correctly, which may include infinite loops and other erratic behavior.

Example

```
1  //% $Id: A25-4-1.cpp 309738 2018-03-01 15:08:00Z michal.szczepankiewicz $
2
3  #include <functional>
4  #include <iostream>
5  #include <set>
6
7  int main(void)
8  {
9      //non-compliant, given predicate does not return false
10     //for equal values
11     std::set<int, std::greater_equal<int>> s{2, 5, 8};
12     auto r = s.equal_range(5);
13     //returns 0
14     std::cout << std::distance(r.first, r.second) << std::endl;
15
16     //compliant, using default std::less<int>
17     std::set<int> s2{2, 5, 8};
18     auto r2 = s2.equal_range(5);
19     //returns 1
20     std::cout << std::distance(r2.first, r2.second) << std::endl;
21
22     return 0;
```

```
}
```

See also

- SEI CERT C++ Coding Standard [10]: CTR57-CPP: Provide a valid ordering predicate
- cppreference.com [16]: C++ concepts: Compare

6.26.5 Random number generation

Rule A26-5-1 (required, implementation, automated)
Pseudorandom numbers shall not be generated using std::rand().

Rationale

Using a pseudo-random sequence of numbers requires that it is generated with good statistical properties. Some implementations of std::rand() function have a comparatively short cycle, as a result the numbers can be predictable. Using functionalities from <random> is recommended instead of using std::rand().

Note: `std::random_shuffle` should not be used, as it is deprecated since C++14 (see A1-1-1) and one of the available overloads is often implemented in terms of `std::rand`.

Example

```
// $Id: A26-5-1.cpp 311495 2018-03-13 13:02:54Z michal.szczepankiewicz $

#include <cstdlib>
#include <cstdint>
#include <ctime>
#include <iostream>
#include <random>

int main()
{
    std::srand(std::time(nullptr));
    int r1 = std::rand() % 100; //non-compliant
    std::cout << "Random value using std::rand(): " << r1 << std::endl;

    std::random_device rd;
    std::default_random_engine eng{rd()};
    std::uniform_int_distribution<int> ud{0, 100};
    int r2 = ud(eng); //compliant
    std::cout << "Random value using std::random_device: " << r2 << std::endl;

    return 0;
}
```

See also

- SEI CERT C++ Coding Standard [10]: MSC50-CPP: Do not use std::rand() for generating pseudorandom numbers.

Rule A26-5-2 (required, implementation, automated)
Random number engines shall not be default-initialized.

Rationale

Using a pseudo-random number generator gives different results that depend on a used seed value. Initializing random number engines by default initializes pseudo-random generator with a *default_seed* constant value. However, this can be not obvious for a developer and can lead to unexpected program behaviour (generating the same random sequences among different program executions).

Exception

For consistent testing purposes it can be convenient to seed the random number engine with a fixed value to get a deterministic sequence, but never within production code where real randomness is required, e.g. for security reasons.

Example

```
1  // $Id: A26-5-2.cpp 311495 2018-03-13 13:02:54Z michal.szczepankiewicz $
2
3  #include <iostream>
4  #include <random>
5
6  int main()
7  {
8      std::random_device rd;
9      std::default_random_engine eng{rd()}; //compliant
10     std::uniform_int_distribution<int> ud{0, 100};
11     int r1 = ud(eng);
12     std::cout << "Random value using std::random_device: " << r1 << std::endl;
13
14
15     std::default_random_engine eng2{};  //non-compliant
16     std::uniform_int_distribution<int> ud2{0, 100};
17     int r2 = ud2(eng);
18     std::cout << "Random value using std::random_device: " << r2 << std::endl;
19
20     return 0;
21 }
```

See also

- SEI CERT C++ Coding Standard [10]: MSC51-CPP: Ensure your random number generator is properly seeded.

6.27 Input/output library - partial

6.27.1 General

Rule M27-0-1 (required, implementation, automated)
The stream input/output library <cstdio> shall not be used.

See MISRA C++ 2008 [7]

Rule A27-0-1 (required, implementation, non-automated)
Inputs from independent components shall be validated.

Rationale

An “attacker” who fully or partially controls the content of an application’s buffer can crash the process, view the content of the stack, view memory content, write to random memory locations or execute code with permissions of the process.

This rule concerns network inputs, as well as inputs that are received from other processes or other software components over IPC or through component APIs.

Note: If more advanced style formatting is required, this can be done using C++ dedicated libraries (e.g. `boost::format` or `libfmt`).

Example

```
1  // $Id: A27-0-1.cpp 311495 2018-03-13 13:02:54Z michal.szczepankiewicz $
2  #include <cstring>
3  #include <cstdint>
4  #include <cstdio>
5  void F1(const char* name)  // name restricted to 256 or fewer characters
6  {
7      static const char format[] = "Name: %s .";
8      size_t len = strlen(name) + sizeof(format);
9      char* msg = new char[len];
10
11     if (msg == nullptr)
12     {
13         // Handle an error
14     }
15
16     std::int32_t ret =
17         snprintf(msg,
18                  len,
19                  format,
20                  name);  // Non-compliant - no additional check for overflows
21
22     if (ret < 0)
```

```
    {
        // Handle an error
    }
    else if (ret >= len)
    {
        // Handle truncated output
    }

    fprintf(stderr, msg);
    delete[] msg;
}
void F2(const char* name)
{
    static const char format[] = "Name: %s .";
    fprintf(stderr, format, name);  // Compliant - untrusted input passed as one
                                    // of the variadic arguments, not as part of
                                    // vulnerable format string
}
void F3(const std::string& name)
{
    //compliant, untrusted input not passed
    //as a part of vulnerable format string
    std::cerr << "Name: " << name;
}
```

See also

- SEI CERT C++ [10]: FIO30-C. Exclude user input from format strings.

Rule A27-0-4 (required, implementation, automated)
C-style strings shall not be used.

Rationale

It is required that an underlying buffer for a C-style string needs to be of sufficient size to hold character data and a null terminator. In addition, a C-style string implies all other disadvantages of built-in arrays (see A18-1-1 in section 6.18.1). Using `std::string` provides correct memory allocation, copying, gradual expansion and iteration. It is self-explanatory in terms of ownership and offers more readable interface.

Example

```
// $Id: A27-0-4.cpp 311495 2018-03-13 13:02:54Z michal.szczepankiewicz $
#include <iostream>
#include <string>
#include <list>

void F1()
{
    std::string string1;
```

```
    std::string string2;
    std::cin >> string1 >> string2;  // Compliant - no buffer overflows
}

std::list<std::string> F2(const std::string& terminator)
{
    std::list<std::string> ret;
    //read a single word until it is different from the given terminator sequence
    for (std::string s; std::cin >> s && s != terminator; )
    {
        ret.push_back(s);
    }
    return ret;
}
```

See also

- C++ Core Guidelines [11]: SL.str.1: Use std::string to own character sequences.

Rule A27-0-2 (advisory, implementation, automated)
A C-style string shall guarantee sufficient space for data and the null terminator.

Rationale

To prevent buffer overflows, it needs to be ensured that the destination is of sufficient size to hold the character data to be copied and the null terminator.

Note that C-style string requires additional space for null character to indicate the end of the string, while the C++ std::basic_string does that implicitly.

Note: This rule is deliberately redundant, in case the rule A27-0-4 is disabled in a project.

Example

```
// $Id: A27-0-2.cpp 289436 2017-10-04 10:45:23Z michal.szczepankiewicz $
#include <iostream>
#include <string>
void F1() noexcept
{
    char buffer[10];
    std::cin >> buffer;  // Non-compliant - this could lead to a buffer overflow
}
void F2() noexcept
{
    std::string string1;
    std::string string2;
    std::cin >> string1 >> string2;  // Compliant - no buffer overflows
}
void F3(std::istream& in) noexcept
```

```
{
    char buffer[32];

    try
    {
        in.read(buffer, sizeof(buffer));
    }

    catch (std::ios_base::failure&)
    {
        // Handle an error
    }

    std::string str(buffer);  // Non-compliant - if 'buffer' is not null
                              // terminated, then constructing std::string leads
                              // to undefined behavior.
}
void F4(std::istream& in) noexcept
{
    char buffer[32];

    try
    {
        in.read(buffer, sizeof(buffer));
    }

    catch (std::ios_base::failure&)
    {
        // Handle an error
    }

    std::string str(buffer, in.gcount());  // Compliant
}
```

See also

- SEI CERT C++ [10]: STR50-CPP. Guarantee that storage for strings has sufficient space for character data and the null terminator.

Rule A27-0-3 (required, implementation, automated)
Alternate input and output operations on a file stream shall not be used without an intervening flush or positioning call.

Rationale

There are following restrictions on reading and writing operations called for an object of class `basic_filebuf<charT, traits>`:

- output shall not be directly followed by input without an intervening call to the fflush function or to a file positioning function (fseek, fsetpos, or rewind).

- input shall not be directly followed by output without an intervening call to a file positioning function, unless the input operation encounters end-of-file.

It is recommended to use a file stream either for output (`std::ofstream`) or input(`std::ifstream`) and not for both in the same context. This avoids the mentioned problem altogether.

Example

```
1  // $Id: A27-0-3.cpp 311495 2018-03-13 13:02:54Z michal.szczepankiewicz $
2
3  #include <fstream>
4  #include <string>
5
6  int main(void)
7  {
8      std::fstream f("testfile");
9
10     f << "Output";
11     std::string str1;
12     f >> str1; // non-compliant
13
14     f << "More output";
15     std::string str2;
16     f.seekg(0, std::ios::beg);
17     f >> str2; //compliant
18
19     return 0;
20 }
```

See also

- SEI CERT C++ Coding Standard [10]: FIO39-C: Do not alternately input and output from a stream without an intervening flush or positioning call
- SEI CERT C++ Coding Standard [10]: FIO50-CPP: Do not alternately input and output from a file stream without an intervening positioning call

7 References

Bibliography

[1] *ISO/IEC 14882:2003, The C++ Standard Incorporating Technical Corrigendum 1*, International Organization for Standardization, 2003.

[2] *ISO/IEC 14882:2011, ISO International Standard ISO/IEC 14882:2011(E) - Programming Language C++*, International Organization of Standardization, 2011.

[3] *ISO/IEC 14882:2014, ISO International Standard ISO/IEC 14882:2014(E) - Programming Language C++*, International Organization for Standardization, 2016.

[4] *ISO 26262-6, Road vehicles - Functional safety - Part 6: Product development at the software level*, International Organization for Standardization, 2011.

[5] *ISO 26262-6, Road vehicles - Functional safety - Part 6: Product development at the software level*, International Organization for Standardization, 2011.

[6] *ISO 26262-8, Road vehicles - Functional safety - Part 8: Supporting processes*, International Organization for Standardization, 2011.

[7] *MISRA C++:2008 Guidelines for the use of the C++ language in critical systems*, The Motor Industry Software Reliability Association, 2008.

[8] *Joint Strike Fighter Air Vehicle C++ Coding Standards for the System Development and Demonstration Program*, Document Number 2RDU00001 Rev C, Lockheed Martin Corporation, 2005.

[9] *High Integrity C++ Coding Standard Version 4.0*, Programming Research Ltd, 2013.

[10] *Software Engineering Institute CERT C++ Coding Standard*, Software Engineering Institute Division at Carnegie Mellon University, 2016.

[11] Bjarne Stroustrup, Herb Sutter, *C++ Core Guidelines*, 2017.

[12] *Google C++ Style Guide*, Google, 2017.

[13] Scott Meyers, *Effective Modern C++*, ISBN: 978-1-491-90399-5, O'Reilly, 2015.

[14] Bjarne Stroustrup, *The C++ Programming Language, Fourth Edition*, ISBN: 978-0-321-56384-2, Addison-Wesley, 2013.

[15] Joshua Bloch, *Effective Java, Second Edition*, ISBN: 978-0321356680, Addison-Wesley, 2008

[16] *cppreference.com, online reference for the C and C++ languages and standard libraries*, 2017

[17] *stackoverflow.com, community of programmers*, 2017

[18] *open-std.org, site holding a number of web pages for groups producing open standards*, 2017

[19] *IEC 61508-3, Functional safety of electrical/electronic/programmable electronic safety-related systems – Annex C, Overview of techniques and measures for achieving software safety integrity*, International Electrotechnical Commission, 2010.

A Allocation of rules to work products

This chapter lists the rules that are allocated to ISO 26262 work products or activities that are impacted by the usage of C++ (in particular for software architectural design specification, software unit design specification, toolchain and others).

The rules listed below can be used as guidelines for peforming those activities or as checklists for performing reviews. For example, it can be used for performing the review of software architectural design specification.

A.1 Rules allocated to architecture

A.2 Rules allocated to design

A.3 Rules allocated to toolchain

A.4 Rules allocated to infrastructure

A.5 Rules allocated to analysis

A.6 Rules allocated to hardware

A.7 Rules allocated to management

A.8 Rules allocated to verification

A.9 Rules allocated to implementation

B Traceability to existing standards

This section demonstrates the traceability of AUTOSAR C++14 rules to existing important C++ coding standards and to ISO 26262.

For each rule, the relation is identified:

1. Identical (only for MISRA C++): the rule text, rationale, exceptions, code example are identical. Only the rule classification can be different. There can be also an additional note with clarifications.
2. Small differences: the content of the rule is included by AUTOSAR C++14 rules with minor differences.
3. Significant differences: the content of the rule is included by AUTOSAR C++14 rules with significant differences.
4. Rejected: the rule in the referred document is rejected by AUTOSAR C++14 guidelines.
5. Not yet analyzed: The rule is not yet analyzed in the current release.
6. Implemented (only for ISO 26262): A clause is implemented by the AUTOSAR C++14 rules.
7. Partially implemented (only for ISO 26262): A clause covered to some extent that is in scope of this document.
8. Not applicable (only for ISO 26262): A clause that is out of scope of this document.

B.1 Traceability to MISRA C++:2008

MISRA C++:2008 [7] is a required prerequisite for readers of the document. MISRA C++:2008 can be purchased over MISRA web store.

The following table demonstrates the traceability to MISRA C++:2008. This is not considered as a reproduction of a part of MISRA C++:2008, but a mean to compare the two standards.

MISRA Rule:	Relation type:	Related rule:	Comment:
0-1-1 (Required) A project shall not contain unreachable code.	1 - Identical	M0-1-1	-
0-1-2 (Required) A project shall not contain infeasible paths.	2 - Small differences	M0-1-2	Note about constexpr functions added.
0-1-3 (Required) A project shall not contain unused variables.	1 - Identical	M0-1-3	-
0-1-4 (Required) A project shall not contain non-volatile POD variables having only one use.	1 - Identical	M0-1-4	-
0-1-5 (Required) A project shall not contain unused type declarations.	3 - Significant differences	A0-1-6	Obligation level changed to “Advisory”.

0-1-6 (Required) A project shall not contain instances of non-volatile variables being given values that are never subsequently used.	2 - Small differences	A0-1-1	Example reworked.
0-1-7 (Required) The value returned by a function having a non-void return type that is not an overloaded operator shall always be used.	2 - Small differences	A0-1-2	Rationale reformulated.
0-1-8 (Required) All functions with void return type shall have external side effect(s).	1 - Identical	M0-1-8	-
0-1-9 (Required) There shall be no dead code.	1 - Identical	M0-1-9	-
0-1-10 (Required) Every defined function shall be called at least once.	3 - Significant differences	M0-1-10, A0-1-2	Rule divided into: (1) Identical rule with obligation level “Advisory”, (2) Rule with obligation level “Required” which applies to static functions and private methods.
0-1-11 (Required) There shall be no unused parameters (named or unnamed) in non-virtual functions.	3 - Significant differences	A0-1-4	Unused parameters are allowed to be unnamed.
0-1-12 (Required) There shall be no unused parameters (named or unnamed) in the set of parameters for a virtual function and all the functions that override it.	3 - Significant differences	A0-1-5	Unused parameters are allowed to be unnamed.
0-2-1 (Required) An object shall not be assigned to an overlapping object.	1 - Identical	M0-2-1	-
0-3-1 (Document) Minimization of run-time failures shall be ensured by the use of at least one of: (a) static analysis tools/techniques; (b) dynamic analysis tools/techniques; (c) explicit coding of checks to handle run-time faults.	1 - Identical	M0-3-1	-
0-3-2 (Required) If a function generates error information, then that error information shall be tested.	1 - Identical	M0-3-2	-
0-4-1 (Document) Use of scaled-integer or fixed-point arithmetic shall be documented.	1 - Identical	M0-4-1	-
0-4-2 (Document) Use of floating-point arithmetic shall be documented.	1 - Identical	M0-4-2	-

0-4-3 (Document) Floating-point implementations shall comply with a defined floating-point standard.	3 - Significant differences	A0-4-1	Specified that floating-point implementations need to comply with IEEE 754 standard.
1-0-1 (Required) All code shall conform to ISO/IEC 14882:2003 “The C++ Standard Incorporating Technical Corrigendum 1”.	2 - Small differences	A1-1-1	Specified that the code shall conform to ISO/IEC 14882:2014.
1-0-2 (Document) Multiple compilers shall only be used if they have a common, defined interface.	1 - Identical	M1-0-2	-
1-0-3 (Document) The implementation of integer division in the chosen compiler shall be determined and documented.	3 - Significant differences	A0-4-2	Specified that the implementation of integer division shall comply with the C++ Language Standard.
2-2-1 (Document) The character set and the corresponding encoding shall be documented.	3 - Significant differences	A2-3-1	
2-3-1 (Required) Trigraphs shall not be used.	2 - Small differences	A2-5-1	All trigraphs listed in rationale. Example extended.
2-5-1 (Advisory) Digraphs should not be used.	3 - Significant differences	A2-5-2	Obligation level changed to “Required”.
2-7-1 (Required) The character sequence /* shall not be used within a C-style comment.	1 - Identical	M2-7-1	-
2-7-2 (Required) Sections of code shall not be commented out using C-style comments.	2 - Small differences	A2-7-1	Commenting-out code sections is not allowed.
2-7-3 (Advisory) Sections of code should not be “commented out” using C++ comments.	2 - Small differences	A2-7-1	Obligation level changed to “Required”. Commenting-out code sections is not allowed.
2-10-1 (Required) Different identifiers shall be typographically unambiguous.	1 - Identical	M2-10-1	-
2-10-2 (Required) Identifiers declared in an inner scope shall not hide an identifier declared in an outer scope.	2 - Small differences	A2-10-1	Added a note to rationale. Example extended.

2-10-3 (Required) A typedef name (including qualification, if any) shall be a unique identifier.	4 - Rejected		This rule is considered as too restrictive.
2-10-4 (Required) A class, union or enum name (including qualification, if any) shall be a unique identifier.	4 - Rejected		This rule is considered as too restrictive.
2-10-5 (Advisory) The identifier name of a non-member object or function with static storage duration should not be reused.	3 - Significant differences	A2-10-4	Obligation level changed to “Required”. Scope of the rule changed.
2-10-6 (Required) If an identifier refers to a type, it shall not also refer to an object or a function in the same scope.	2 - Small differences	A2-10-6	
2-13-1 (Required) Only those escape sequences that are defined in ISO/IEC 14882:2003 shall be used.	2 - Small differences	A2-13-1	Standard changed to ISO/IEC 14882:2014.
2-13-2 (Required) Octal constants (other than zero) and octal escape sequences (other than “\0”) shall not be used.	1 - Identical	M2-13-2	-
2-13-3 (Required) A “U” suffix shall be applied to all octal or hexadecimal integer literals of unsigned type.	1 - Identical	M2-13-3	-
2-13-4 (Required) Literal suffixes shall be upper case.	1 - Identical	M2-13-4	-
2-13-5 (Required) Narrow and wide string literals shall not be concatenated.	2 - Small differences	A2-13-2	Example extended.
3-1-1 (Required) It shall be possible to include any header file in multiple translation units without violating the One Definition Rule.	3 - Significant differences	A3-1-1	Rationale reformulated. Example extended.
3-1-2 (Required) Functions shall not be declared at block scope.	1 - Identical	M3-1-2	-
3-1-3 (Required) When an array is declared, its size shall either be stated explicitly or defined implicitly by initialization.	2 - Small differences	A3-1-4	Specified that this rule applies to arrays with external linkage only.
3-2-1 (Required) All declarations of an object or function shall have compatible types.	1 - Identical	M3-2-1	-
3-2-2 (Required) The One Definition Rule shall not be violated.	1 - Identical	M3-2-2	-
3-2-3 (Required) A type, object or function that is used in multiple translation units shall be declared in one and only one file.	1 - Identical	M3-2-3	-

3-2-4 (Required) An identifier with external linkage shall have exactly one definition.	1 - Identical	M3-2-4	-
3-3-1 (Required) Objects or functions with external linkage shall be declared in a header file.	2 - Small differences	A3-3-1	Added a note to rationale. Example extended.
3-3-2 (Required) If a function has internal linkage then all re-declarations shall include the static storage class specifier.	1 - Identical	M3-3-2	-
3-4-1 (Required) An identifier declared to be an object or type shall be defined in a block that minimizes its visibility.	1 - Identical	M3-4-1	-
3-9-1 (Required) The types used for an object, a function return type, or a function parameter shall be token-for-token identical in all declarations and re-declarations.	1 - Identical	M3-9-1	-
3-9-2 (Advisory) typedefs that indicate size and signedness should be used in place of the basic numerical types.	3 - Significant differences	M3-9-1	Rule title and rationale reformulated to use types from <cstdint> header file. All types that should be used were listed. Example changed.
3-9-3 (Required) The underlying bit representations of floating-point values shall not be used.	1 - Identical	M3-9-3	-
4-5-1 (Required) Expressions with type bool shall not be used as operands to built-in operators other than the assignment operator =, the logical operators &&, ||, !, the equality operators == and !=, the unary & operator, and the conditional operator.	1 - Identical	M4-5-1	-
4-5-2 (Required) Expressions with type enum shall not be used as operands to built-in operators other than the subscript operator [ ], the assignment operator =, the equality operators == and !=, the unary & operator, and the relational operators <, <=, >, >=.	3 - Significant differences	A4-5-1	Changed the rule so it applies to enum classes too. Rationale reformulated. Example extended.
4-5-3 (Required) Expressions with type (plain) char and wchar_t shall not be used as operands to built-in operators other than the assignment operator =, the equality operators == and !=, and the unary & operator.	1 - Identical	M4-5-3	-
4-10-1 (Required) NULL shall not be used as an integer value.	1 - Identical	M4-10-1	-

4-10-2 (Required) Literal zero (0) shall not be used as the null-pointer-constant.	1 - Identical	M4-10-2	-
5-0-1 (Required) The value of an expression shall be the same under any order of evaluation that the standard permits.	1 - Identical	A5-0-1	Example rewritten to compile with C++ compiler
5-0-2 (Advisory) Limited dependence should be placed on C++ operator precedence rules in expressions.	1 - Identical	M5-0-2	-
5-0-3 (Required) A cvalue expression shall not be implicitly converted to a different underlying type.	1 - Identical	M5-0-3	-
5-0-4 (Required) An implicit integral conversion shall not change the signedness of the underlying type.	1 - Identical	M5-0-4	-
5-0-5 (Required) There shall be no implicit floating-integral conversions.	1 - Identical	M5-0-5	-
5-0-6 (Required) An implicit integral or floating-point conversion shall not reduce the size of the underlying type.	1 - Identical	M5-0-6	-
5-0-7 (Required) There shall be no explicit floating-integral conversions of a cvalue expression.	1 - Identical	M5-0-7	-
5-0-8 (Required) An explicit integral or floating-point conversion shall not increase the size of the underlying type of a cvalue expression.	1 - Identical	M5-0-8	-
5-0-9 (Required) An explicit integral conversion shall not change the signedness of the underlying type of a cvalue expression.	1 - Identical	M5-0-9	-
5-0-10 (Required) If the bitwise operators and « are applied to an operand with an underlying type of unsigned char or unsigned short, the result shall be immediately cast to the underlying type of the operand.	1 - Identical	M5-0-10	-
5-0-11 (Required) The plain char type shall only be used for the storage and use of character values.	1 - Identical	M5-0-11	-
5-0-12 (Required) signed char and unsigned char type shall only be used for the storage and use of numeric values.	1 - Identical	M5-0-12	-
5-0-13 (Required) The condition of an if-statement and the condition of an iteration statement shall have type bool.	2 - Small differences	A5-0-2	Example extended.
5-0-14 (Required) The first operand of a conditional-operator shall have type bool.	1 - Identical	M5-0-14	-
5-0-15 (Required) Array indexing shall be the only form of pointer arithmetic.	1 - Identical	M5-0-15	-

5-0-16 (Required) A pointer operand and any pointer resulting from pointer arithmetic using that operand shall both address elements of the same array.	1 - Identical	M5-0-16	-
5-0-17 (Required) Subtraction between pointers shall only be applied to pointers that address elements of the same array.	1 - Identical	M5-0-17	-
5-0-18 (Required) >, >=, <, <= shall not be applied to objects of pointer type, except where they point to the same array.	1 - Identical	M5-0-18	-
5-0-19 (Required) The declaration of objects shall contain no more than two levels of pointer indirection.	2 - Small differences	A5-0-3	Example changed - typedef replaced with using.
5-0-20 (Required) Non-constant operands to a binary bitwise operator shall have the same underlying type.	1 - Identical	M5-0-20	-
5-0-21 (Required) Bitwise operators shall only be applied to operands of unsigned underlying type.	1 - Identical	M5-0-21	-
5-2-1 (Required) Each operand of a logical && or || shall be a postfix expression.	3 - Significant differences	A5-2-6	Rule formulation simplified.
5-2-2 (Required) A pointer to a virtual base class shall only be cast to a pointer to a derived class by means of dynamic_cast.	1 - Identical	M5-2-2	-
5-2-3 (Advisory) Casts from a base class to a derived class should not be performed on polymorphic types.	1 - Identical	M5-2-3	-
5-2-4 (Required) C-style casts (other than void casts) and functional notation casts (other than explicit constructor calls) shall not be used.	3 - Significant differences	A5-2-2	Rule title and rationale reformulated, detailed explanation and possible alternatives added. Example reworked.
5-2-5 (Required) A cast shall not remove any const or volatile qualification from the type of a pointer or reference.	2 - Small differences	A5-2-3	Added a note to rationale. Example reworked.
5-2-6 (Required) A cast shall not convert a pointer to a function to any other pointer type, including a pointer to function type.	1 - Identical	M5-2-6	-

5-2-7 (Required) An object with pointer type shall not be converted to an unrelated pointer type, either directly or indirectly.	3 - Significant differences	A5-2-4	Rule title and rationale reformulated to prohibit reinterpret_cast usage. Example reworked.
5-2-8 (Required) An object with integer type or pointer to void type shall not be converted to an object with pointer type.	1 - Identical	M5-2-8	-
5-2-9 (Advisory) A cast shall not convert a pointer type to an integral type.	2 - Small differences	M5-2-9	Obligation level changed to “Required”.
5-2-10 (Advisory) The increment (++) and decrement (–) operators shall not be mixed with other operators in an expression.	2 - Small differences	M5-2-10	Obligation level changed to “Required”.
5-2-11 (Required) The comma operator, && operator and the operator shall not be overloaded.	1 - Identical	M5-2-11	-
5-2-12 (Required) An identifier with array type passed as a function argument shall not decay to a pointer.	1 - Identical	M5-2-12	-
5-3-1 (Required) Each operand of the ! operator, the logical && or the logical ||operators shall have type bool.	1 - Identical	M5-3-1	-
5-3-2 (Required) The unary minus operator shall not be applied to an expression whose underlying type is unsigned.	1 - Identical	M5-3-2	-
5-3-3 (Required) The unary & operator shall not be overloaded.	1 - Identical	M5-3-3	-
5-3-4 (Required) Evaluation of the operand to the sizeof operator shall not contain side effects.	1 - Identical	M5-3-4	-
5-8-1 (Required) The right hand operand of a shift operator shall lie between zero and one less than the width in bits of the underlying type of the left hand operand.	1 - Identical	M5-8-1	-
5-14-1 (Required) The right hand operand of a logical && or ||operator shall not contain side effects.	1 - Identical	M5-14-1	-
5-17-1 (Required) The semantic equivalence between a binary operator and its assignment operator form shall be preserved.	1 - Identical	M5-17-1	-
5-18-1 (Required) The comma operator shall not be used.	1 - Identical	M5-18-1	-

5-19-1 (Required) Evaluation of constant unsigned integer expressions shall not lead to wrap-around.	2 - Small differences	M5-19-1	Obligation level changed to “Required”.
6-2-1 (Required) Assignment operators shall not be used in subexpressions.	1 - Identical	M6-2-1	-
6-2-2 (Required) Floating-point expressions shall not be directly or indirectly tested for equality or inequality.	1 - Identical	M6-2-2	-
6-2-3 (Required) Before preprocessing, a null statement shall only occur on a line by itself; it may be followed by a comment, provided that the first character following the null statement is a white-space character.	1 - Identical	M6-2-3	-
6- 3-1 (Required) The statement forming the body of a switch, while, do ... while or for statement shall be a compound statement.	1 - Identical	M6-3-1	-
6-4-1 (Required) An if (condition) construct shall be followed by a compound statement. The else keyword shall be followed by either a compound statement, or another if statement.	1 - Identical	M6-4-1	-
6-4-2 (Required) All if ... else if constructs shall be terminated with an else clause.	1 - Identical	M6-4-2	-
6-4- 3 (Required) A switch statement shall be a well-formed switch statement.	1 - Identical	M6-4-3	-
6-4-4 (Required) A switch-label shall only be used when the most closely-enclosing compound statement is the body of a switch statement.	1 - Identical	M6-4-4	-
6-4-5 (Required) An unconditional throw or break statement shall terminate every non-empty switch clause.	1 - Identical	M6-4-5	-
6-4-6 (Required) The final clause of a switch statement shall be the default-clause.	1 - Identical	M6-4-6	-
6-4-7 (Required) The condition of a switch statement shall not have bool type.	1 - Identical	M6-4-7	-
6- 4-8 (Required) Every switch statement shall have at least one case-clause.	3 - Significant differences	A6-4-1	Rule reformulated. Example reworked.

6-5-1 (Required) A for loop shall contain a single loop-counter which shall not have floating type.	2 - Small differences	A6-5-2	Additional note about floating types added. Rule extended.
6-5-2 (Required) If loop-counter is not modified by – or ++, then, within condition, the loop-counter shall only be used as an operand to <=, <, > or >=.	1 - Identical	M6-5-2	-
6-5-3 (Required) The loop-counter shall not be modified within condition or statement.	1 - Identical	M6-5-3	-
6-5-4 (Required) The loop-counter shall be modified by one of: –, ++, -=n, or +=n; where n remains constant for the duration of the loop.	1 - Identical	M6-5-4	-
6-5-5 (Required) A loop-control-variable other than the loop-counter shall not be modified within condition or expression.	1 - Identical	M6-5-5	-
6-5-6 (Required) A loop-control-variable other than the loop-counter which is modified in statement shall have type bool.	1 - Identical	M6-5-6	-
6-6-1 (Required) Any label referenced by a goto statement shall be declared in the same block, or in a block enclosing the goto statement.	1 - Identical	M6-6-1	-
6-6-2 (Required) The goto statement shall jump to a label declared later in the same function body.	1 - Identical	M6-6-2	-
6-6-3 (Required) The continue statement shall only be used within a well-formed for loop.	1 - Identical	M6-6-3	-
6-6-4 (Required) For any iteration statement there shall be no more than one break or goto statement used for loop termination.	4 - Rejected		The goto statement shall not be used, see: A6-6-1. There can be more than one break in an iteration statement.
6-6-5 (Required) A function shall have a single point of exit at the end of the function.	4 - Rejected		See `Single-point-of-exit`.
7-1-1 (Required) A variable which is not modified shall be const qualified.	3 - Significant differences	A7-1-1, A7-1-2	constexpr and const specifiers are recommended.

7-1-2 (Required) A pointer or reference parameter in a function shall be declared as pointer to const or reference to const if the corresponding object is not modified.	1 - Identical	M7-1-2	-
7-2-1 (Required) An expression with enum underlying type shall only have values corresponding to the enumerators of the enumeration.	2 - Small differences	A7-1-2	Example extended.
7-3-1 (Required) The global namespace shall only contain main, namespace declarations and extern "C" declarations.	1 - Identical	M7-3-1	-
7-3-2 (Required) The identifier main shall not be used for a function other than the global function main.	1 - Identical	M7-3-2	-
7-3-3 (Required) There shall be no unnamed namespaces in header files.	1 - Identical	M7-3-3	-
7-3-4 (Required) Using-directives shall not be used.	1 - Identical	M7-3-4	-
7-3-5 (Required) Multiple declarations for an identifier in the same namespace shall not straddle a using-declaration for that identifier.	2 - Small differences	A7-3-1	
7-3-6 (Required) using-directives and using-declarations (excluding class scope or function scope using-declarations) shall not be used in header files.	1 - Identical	M7-3-6	-
7-4- 1 (Document) All usage of assembler shall be documented.	1 - Identical	M7-4-1	-
7- 4-2 (Required) Assembler instructions shall only be introduced using the asm declaration.	1 - Identical	M7-4-2	-
7-4-3 (Required) Assembly language shall be encapsulated and isolated.	1 - Identical	M7-4-3	-
7-5-1 (Required) A function shall not return a reference or a pointer to an automatic variable (including parameters), defined within the function.	1 - Identical	M7-5-1	-
7-5-2 (Required) The address of an object with automatic storage shall not be assigned to another object that may persist after the first object has ceased to exist.	2 - Small differences	M7-5-2	Added a note saying that the rule applies to std::unique_ptr, std::shared_ptr and std::weak_ptr too.

7-5-3 (Required) A function shall not return a reference or a pointer to a parameter that is passed by reference or const reference.	3 - Significant differences	A7-5-2	Rule reformulated so it is allowed to return a reference or a pointer to non-const reference parameter. Rationale reformulated. Example reworked.
7-5-4 (Advisory) Functions should not call themselves, either directly or indirectly.	2 - Small differences	A7-5-1	Obligation level changed to “Required”. Example reworked.
8-0-1 (Required) An init-declarator-list or a member-declarator-list shall consist of a single init-declarator or member-declarator respectively.	1 - Identical	M8-0-1	-
8-3-1 (Required) Parameters in an overriding virtual function shall either use the same default arguments as the function they override, or else shall not specify any default arguments.	1 - Identical	M8-3-1	-
8-4-1 (Required) Functions shall not be defined using the ellipsis notation.	3 - Significant differences	A8-4-1	Rationale reformulated. Added a note that variadic templates should be used instead. Example extended.
8-4-2 (Required) The identifiers used for the parameters in a re-declaration of a function shall be identical to those in the declaration.	1 - Identical	M8-4-2	-
8-4-3 (Required) All exit paths from a function with non-void return type shall have an explicit return statement with an expression.	2 - Small differences	A8-4-2	Rule reformulated so it applies to void return type functions. Example reworked so there is no throwing an exception of type int.
8-4-4 (Required) A function identifier shall either be used to call the function or it shall be preceded by &.	1 - Identical	M8-4-4	-
8-5-1 (Required) All variables shall have a defined value before they are used.	2 - Small differences	A8-5-0	Rule reworded to also include objects with dynamic storage.

8-5-2 (Required) Braces shall be used to indicate and match the structure in the non-zero initialization of arrays and structures.	1 - Identical	M8-5-2	-
8-5-3 (Required) In an enumerator list, the = construct shall not be used to explicitly initialize members other than the first, unless all items are explicitly initialized.	3 - Significant differences	A7-2-4	Rule and rationale reformulated. Example reworked.
9-3- 1 (Required) const member functions shall not return non-const pointers or references to class-data.	1 - Identical	M9-3-1	-
9-3-2 (Required) Member functions shall not return non-const handles to class-data.	2 - Small differences	A9-3-1	Explanation improved. Example reworked.
9-3-3 (Required) If a member function can be made static then it shall be made static, otherwise if it can be made const then it shall be made const.	1 - Identical	M9-3-3	-
9-5-1 (Required) Unions shall not be used.	2 - Small differences	A9-5-1	-
9-6-1 (Required) When the absolute positioning of bits representing a bit-field is required, then the behavior and packing of bit-fields shall be documented.	1 - Identical	M9-6-1	-
9-6-2 (Required) Bit-fields shall be either bool type or an explicitly unsigned or signed integral type.	3 - Significant differences	A9-6-1	Only types with a defined size are allowed to be used for bit-fields, also see A9-6-2.
9-6-3 (Required) Bit-fields shall not have enum type.	4 - Rejected	-	Permitted types changed. New rule introduced: A9-6-1.
9-6-4 (Required) Named bit-fields with signed integer type shall have a length of more than one bit.	1 - Identical	M9-6-4	
10-1-1 (Advisory) Classes should not be derived from virtual bases	1 - Identical	M10-1-1	-
10-1-2 (Required) A base class shall only be declared virtual if it is used in a diamond hierarchy.	1 - Identical	M10-1-2	
10-1-3 (Required) An accessible base class shall not be both virtual and non-virtual in the same hierarchy.	1 - Identical	M10-1-3	-
10-2-1 (Advisory) All accessible entity names within a multiple inheritance hierarchy should be unique.	1 - Identical	M10-2-1	-

10-3-1 (Required) There shall be no more than one definition of each virtual function on each path through the inheritance hierarchy.	4 - Rejected	-	Rule already covered by A10-1-1.
10-3-2 (Required) Each overriding virtual function shall be declared with the virtual keyword.	3 - Significant differences	A10-3-2	Rule and rationale reformulated so the override specifier should be used instead of virtual keyword. Example reworked.
10-3-3 (Required) A virtual function shall only be overridden by a pure virtual function if it is itself declared as pure virtual.	1 - Identical	M10-3-3	-
11-0-1 (Required) Member data in non-POD class types shall be private.	1 - Identical	M11-0-1	-
12-1-1 (Required) An object's dynamic type shall not be used from the body of its constructor or destructor.	1 - Identical	M12-1-1	-
12-1-2 (Advisory) All constructors of a class should explicitly call a constructor for all of its immediate base classes and all virtual base classes.	3 - Significant differences	A12-1-1	Obligation level changed to "Required". Rule reformulated to cover non-static class data members. Rationale reformulated. Example reworked.
12-1-3 (Required) All constructors that are callable with a single argument of fundamental type shall be declared explicit.	2 - Small differences	A12-1-4	Example reworked.
12-8-1 (Required) A copy constructor shall only initialize its base classes and the non-static members of the class of which it is a member.	3 - Significant differences	A12-8-1	Rule reformulated to cover move constructors, too. Rationale reformulated. Example reworked.
12-8-2 (Required) The copy assignment operator shall be declared protected or private in an abstract class.	3 - Significant differences	A12-8-6	Rule reformulated to cover move assignment operators and all base classes. Rationale reformulated. Example reworked.

12- 8-2 (Required) The copy assignment operator shall be declared protected or private in an abstract class.	3 - Significant differences	A12-8-1	Rule reformulated to cover move constructors, too. Rationale reformulated. Example reworked.
14-5-1 (Required) A non-member generic function shall only be declared in a namespace that is not an associated namespace.	3 - Significant differences	A14-5-3	Changed the scope of the rule from all generic functions to operators only, as most problematic case. Changed severity to Advisory.
14-5-2 (Required) A copy constructor shall be declared when there is a template constructor with a single parameter that is a generic parameter.	3 - Significant differences	A14-5-1	Avoids the ambiguity by requiring the template parameter to be constrained to never match a copy/move constructor.
14-5-3 (Required) A copy assignment operator shall be declared when there is a template assignment operator with a parameter that is a generic parameter.	1 - Identical	M14-5-3	-
14-6-1 (Required) In a class template with a dependent base, any name that may be found in that dependent base shall be referred to using a qualified-id or this->.	1 - Identical	M14-6-1	-
14-6-2 (Required) The function chosen by overload resolution shall resolve to a function declared previously in the translation unit.	4 - Rejected	-	Usage of the ADL functionality is allowed. It is also used in STL for overloaded operators lookup in e.g. out streams, STL containers.
14-7-1 (Required) All class templates, function templates, class template member functions and class template static members shall be instantiated at least once.	4 - Rejected	-	It is allowed to not use all of the public methods of a class.

14-7-2 (Required) For any given template specialization, an explicit instantiation of the template with the template-arguments used in the specialization shall not render the program ill-formed.	3 - Significant differences	A14-7-1	Rule reformulated to explicitly state what is required. Example reworked.
14-7-3 (Required) All partial and explicit specializations for a template shall be declared in the same file as the declaration of their primary template.	3 - Significant differences	A14-7-2	Allowed to declare in a header file that declares user-defined type, for which the specialization is declared.
14-8-1 (Required) Overloaded function templates shall not be explicitly specialized.	3 - Significant differences	A14-8-2	Function templates specialization is forbidden.
14-8-2 (Advisory) The viable function set for a function call should either contain no function specializations, or only contain function specializations.	3 - Small differences	A14-8-2	Function templates specialization is forbidden.
15-0-1 (Document) Exceptions shall only be used for error handling.	3 - Significant differences	A15-0-1	Rule reformulated, example significantly extended.
15-0-2 (Advisory) An exception object should not have pointer type.	3 - Significant differences	A15-1-2	Obligation level changed, rule reformulated.
15-0-3 (Required) Control shall not be transferred into a try or catch block using a goto or a switch statement.	1 - Identical	M15-0-3	-
15-1-1 (Required) The assignment-expression of a throw statement shall not itself cause an exception.	1 - Identical	M15-1-1	-
15-1-2 (Required) NULL shall not be thrown explicitly.	1 - Identical	M15-1-2	-
15-1-3 (Required) An empty throw (throw;) shall only be used in the compound-statement of a catch handler.	1 - Identical	M15-1-3	-
15-3-1 (Required) Exceptions shall be raised only after start-up and before termination of the program.	1 - Identical	M15-3-1	-
15-3-2 (Advisory) There should be at least one exception handler to catch all otherwise unhandled exceptions.	2 - Small differences	A15-3-3	Obligation level changed. Rule extended to cover multi-threading.

15-3-3 (Required) Handlers of a function-try-block implementation of a class constructor or destructor shall not reference non-static members from this class or its bases.	1 - Identical	M15-3-3	-
15-3-4 (Required) Each exception explicitly thrown in the code shall have a handler of a compatible type in all call paths that could lead to that point.	1 - Identical	M15-3-4	-
15- 3-5 (Required) A class type exception shall always be caught by reference.	2 - Small differences	A15-3-5	Possibility to catch by const reference added
15-3-6 (Required) Where multiple handlers are provided in a single try-catch statement or function-try-block for a derived class and some or all of its bases, the handlers shall be ordered most-derived to base class.	1 - Identical	M15-3-6	-
15-3-7 (Required) Where multiple handlers are provided in a single try-catch statement or function-try-block, any ellipsis (catch-all) handler shall occur last.	1 - Identical	M15-3-7	-
15-4-1 (Required) If a function is declared with an exception-specification, then all declarations of the same function (in other translation units) shall be declared with the same set of type-ids.	3 - Significant differences	A15-4-3	Dynamic exception specification was prohibited. Rule was reformulated in terms of the noexcept specification.
15-5-1 (Required) A class destructor shall not exit with an exception.	3 - Significant differences	A15-5-1	Rule significantly extended with other special functions and operators.
15-5-2 (Required) Where a function's declaration includes an exception-specification, the function shall only be capable of throwing exceptions of the indicated type(s).	3 - Significant differences	A15-4-2	Dynamic exception specification was prohibited. Rule was reformulated in terms of the noexcept specification.
15-5-3 (Required) The std::terminate() function shall not be called implicitly.	2 - Small differences	A15-5-3	Rationale and example extended.
16-0-1 (Required) #include directives in a file shall only be preceded by other preprocessor directives or comments.	1 - Identical	M16-0-1	-

16-0-2 (Required) Macros shall only be #define'd or #undef'd in the global namespace.	1 - Identical	M16-0-2	-
16-0-3 (Required) #undef shall not be used.	4 - Rejected	-	The rule replaced with global rule: A16-0-1.
16-0-4 (Required) Function-like macros shall not be defined.	4 - Rejected	-	The rule replaced with global rule: A16-0-1.
16-0-5 (Required) Arguments to a function-like macro shall not contain tokens that look like preprocessing directives.	1 - Identical	M16-0-5	-
16-0-6 (Required) In the definition of a function-like macro, each instance of a parameter shall be enclosed in parentheses, unless it is used as the operand of # or ##.	1 - Identical	M16-0-6	-
16-0-7 (Required) Undefined macro identifiers shall not be used in #if or #elif preprocessor directives, except as operands to the defined operator.	1 - Identical	M16-0-7	-
16-0-8 (Required) If the # token appears as the first token on a line, then it shall be immediately followed by a pre-processing token.	1 - Identical	M16-0-8	-
16-1- 1 (Required) The defined preprocessor operator shall only be used in one of the two standard forms.	1 - Identical	M16-1-1	-
16-1-2 (Required) All #else, #elif and #endif pre-processor directives shall reside in the same file as the #if or #ifdef directive to which they are related.	1 - Identical	M16-1-2	-
16-2-1 (Required) The pre-processor shall only be used for file inclusion and include guards.	4 - Rejected	-	The rule replaced with global rule: A16-0-1.
16-2-2 (Required) C++ macros shall only be used for include guards, type qualifiers, or storage class specifiers.	4 - Rejected	-	The rule replaced with global rule: A16-0-1.
16-2-3 (Required) Include guards shall be provided	1 - Identical	M16-2-3	-

16-2-4 (Required) The ', ", /* or // characters shall not occur in a header file name.	2 - Small differences	A16-2-1	Merged with MISRA Rule 16-2-5.
16-2-5 (Advisory) The character \should not occur in a header file name.	2 - Small differences	A16-2-1	Obligation level changed to "Required". Merged with MISRA Rule 16-2-4.
16-2-6 (Required) The #include directive shall be followed by either a <filename> or "filename" sequence.	4 - Rejected	-	These are the only forms allowed by the C++ Language Standard; No need for a new rule.
16-3-1 (Required) There shall be at most one occurrence of the # or ## operators in a single macro definition.	1 - Identical	M16-3-1	-
16-3-2 (Advisory) The # and ## operators should not be used.	1 - Identical	M16-3-2	-
16-6-1 (Required) All uses of the #pragma directive shall be documented.	4 - Rejected	-	The #pragma directive shall not be used, see: A16-7-1.
17-0-1 (Required) Reserved identifiers, macros and functions in the standard library shall not be defined, redefined or undefined.	2 - Small differences	A17-0-1	Example extended.
17-0-2 (Required) The names of standard library macros and objects shall not be reused.	1 - Identical	M17-0-2	-
17-0-3 (Required) The names of standard library functions shall not be overridden.	1 - Identical	M17-0-3	-
17-0-4 (Required) All library code shall conform to MISRA C++.	4 - Rejected	-	The rule replaced with A17-0-2 saying that all code shall conform to AUTOSAR C++14 Coding Guidelines.
17-0-5 (Required) The setjmp macro and the longjmp function shall not be used.	1 - Identical	M17-0-5	-
18-0-1 (Required) The C library shall not be used.	2 - Small differences	A18-0-1	Rule reformulated.
18-0-2 (Required) The library functions atof, atoi and atol from library <cstdlib> shall not be used.	2 - Small differences	A18-0-2	

18-0-3 (Required) The library functions abort, exit, getenv and system from library <cstdlib> shall not be used.	1 - Identical	M18-0-3	-
18-0-4 (Required) The time handling functions of library <ctime> shall not be used.	1 - Identical	M18-0-4	-
18-0-5 (Required) The unbounded functions of library <cstring> shall not be used.	1 - Identical	M18-0-5	-
18-2-1 (Required) The macro offsetof shall not be used.	1 - Identical	M18-2-1	-
18-4-1 (Required) Dynamic heap memory allocation shall not be used.	4 - Rejected	-	Dynamic heap memory allocation usage is allowed conditionally, see: A18-5-1, A18-5-2, A18-5-3.
18-7-1 (Required) The signal handling facilities of <csignal> shall not be used.	1 - Identical	M18-7-1	-
19-3-1 (Required) The error indicator errno shall not be used.	1 - Identical	M19-3-1	-
27-0-1 (Required) The stream input/output library <cstdio> shall not be used.	1 - Identical	M27-0-1	-

Table B.1: MISRA C++

B.2 Traceability to HIC++ v4.0

The following table demonstrates the traceability to High Integrity C++ Coding Standard Version 4.0 [9]. This is not considered as a reproduction, but a mean to compare the two standards.

This document complies with the conditions of use of HIC++ v4.0, as any rule in this document that is based on HIC++ v4.0 refers to the related HIC++ v4.0 rule.

HIC++ Rule:	Relation type:	Related rule:	Comment:
1.1.1 Ensure that code complies with the 2011 ISO C++ Language Standard.	2 - Small differences	A1-1-1	Specified that the code shall conform to ISO/IEC 14882:2014
1.2.1 Ensure that all statements are reachable.	2 - Small differences	M0-1-1	
1.2.2 Ensure that no expression or sub-expression is redundant.	2 - Small differences	M0-1-9	
1.3.1 Do not use the increment operator (++) on a variable of type bool	2 - Small differences	M4-5-1	
1.3.2 Do not use the register keyword.	2 - Small differences	A7-1-4	

1.3.3 Do not use the C Standard Library .h headers	2 - Small differences	A18-0-1	
1.3.4 Do not use deprecated STL library features	2 - Small differences	A1-1-1, A18-1-3, A18-9-1	
1.3.5 Do not use throw exception specifications.	2 - Small differences	A15-4-1	
2.1.1 Do not use tab characters in source files.	4 - Rejected		AUTOSAR C++ Coding Guidelines does not introduce rules related to coding style or naming convention.
2.2.1 Do not use digraphs or trigraphs.	2 - Small differences	A2-5-1, A2-5-2	
2.3.1 Do not use the C comment delimiters /* ... */.	2 - Small differences	M2-7-1	
2.3.2 Do not comment out code.	2 - Small differences	A2-7-2	
2.4.1 Ensure that each identifier is distinct from any other visible identifier.	2 - Small differences	M2-10-1	
2.5.1 Do not concatenate strings with different encoding prefixes.	2 - Small differences	A2-13-2	
2.5.2 Do not use octal constants (other than zero).	2 - Small differences	M2-13-2	
2.5.3 Use nullptr for the null pointer constant.	2 - Small differences	A4-10-1	
3.1.1 Do not hide declarations.	2 - Small differences	A2-10-1, A2-10-6	
3.2.1 Do not declare functions at block scope.	2 - Small differences	M3-1-2	
3.3.1 Do not use variables with static storage duration.	3 - Significant differences	A3-3-2	Limited to constant-initialized objects only.
3.4.1 Do not return a reference or a pointer to an automatic variable defined within the function.	2 - Small differences	M7-5-1	
3.4.2 Do not assign the address of a variable to a pointer with a greater lifetime.	2 - Small differences	M7-5-2	
3.4.3 Use RAII for resources.	4 - Rejected		AUTOSAR C++ Coding Guidelines does not define rules for coding patterns. Note that usage of RAII is recommended, see: A15-1-4.

3.5.1 Do not make any assumptions about the internal representation of a value or object.	2 - Small differences	A3-9-1, M3-9-3, M5-0-15, M5-0-21, A9-5-1, M18-2-1	
4.1.1 Ensure that a function argument does not undergo an array-to-pointer conversion.	2 - Small differences	M5-2-12	
4.2.1 Ensure that the U suffix is applied to a literal used in a context requiring an unsigned integral expression.	2 - Small differences	M2-13-2	
4.2.2 Ensure that data loss does not demonstrably occur in an integral expression.	2 - Small differences	A4-7-1, M5-0-4, M5-0-6, M5-0-9	
4.3.1 Do not convert an expression of wider floating point type to a narrower floating point type.	2 - Small differences	A4-7-1, M5-0-6	
4.4.1 Do not convert floating values to integral types except through use of standard library functions.	4 - Rejected		Rules that are related: M5-0-3, M5-0-5, M5-0-6, M5-0-7,
5.1.1 Use symbolic names instead of literal values in code.	2 - Small differences	A5-1-1	
5.1.2 Do not rely on the sequence of evaluation within an expression.	2 - Small differences	A5-0-1	
5.1.3 Use parentheses in expressions to specify the intent of the expression.	2 - Small differences	A5-0-1, A5-2-6, M5-2-10,	
5.1.4 Do not capture variables implicitly in a lambda.	2 - Small differences	A5-1-2	
5.1.5 Include a (possibly empty) parameter list in every lambda expression.	2 - Small differences	A5-1-3	
5.1.6 Do not code side effects into the right-hand operands of: &&, ||, sizeof, typeid or a function passed to condition_variable::wait.	3 - Significant differences	A5-3-1, M5-3-4, M5-14-1	The condition_variable::wait is not yet covered, this will be addressed in future when C++ libraries are analyzed.
5.2.1 Ensure that pointer or array access is demonstrably within bounds of a valid object.	2 - Small differences	A5-2-5	
5.2.2 Ensure that functions do not call themselves, either directly or indirectly.	2 - Small differences	A7-5-2	
5.3.1 Do not apply unary minus to operands of unsigned type.	2 - Small differences	M5-3-2	

5.3.2 Allocate memory using new and release it using delete.	2 - Small differences	A18-5-1	Note that operators new and delete shall not be used explicitly, see: A18-5-2.
5.3.3 Ensure that the form of delete matches the form of new used to allocate the memory.	2 - Small differences	A18-5-3	Note that operators new and delete shall not be used explicitly, see: A18-5-2.
5.4.1 Only use casting forms: static_cast (excl. void*), dynamic_cast or explicit constructor call.	2 - Small differences	A5-2-1, A5-2-2, A5-2-3, A5-2-4	
5.4.2 Do not cast an expression to an enumeration type.	4 - Rejected		It is allowed to cast an expression to an enumeration type, but an expression shall have a value that corresponds to an enumerator of the enumeration, see: A7-2-1.
5.4.3 Do not convert from a base class to a derived class.	3 - Small differences	M5-2-2, M5-2-3, A5-2-1	Note that the dynamic_cast is unsuitable for use with real-time systems.
5.5.1 Ensure that the right hand operand of the division or remainder operators is demonstrably non-zero.	2 - Small differences	A5-6-1	
5.6.1 Do not use bitwise operators with signed operands.	2 - Small differences	M5-0-21	
5.7.1 Do not write code that expects floating point calculations to yield exact results.	2 - Small differences	M6-2-2	
5.7.2 Ensure that a pointer to member that is a virtual function is only compared (==) with nullptr.	2 - Small differences	A5-10-1	
5.8.1 Do not use the conditional operator (?:) as a sub-expression.	2 - Small differences	A5-16-1	
6.1.1 Enclose the body of a selection or an iteration statement in a compound statement.	2 - Small differences	M6-3-1, M6-4-1	
6.1.2 Explicitly cover all paths through multi-way selection statements.	2 - Small differences	M6-4-2	
6.1.3 Ensure that a non-empty case statement block does not fall through to the next label.	2 - Small differences	M6-4-5	

6.1.4 Ensure that a switch statement has at least two case labels, distinct from the default label.	2 - Small differences	A6-4-1	
6.2.1 Implement a loop that only uses element values as a range-based loop.	2 - Small differences	A6-5-1	
6.2.2 Ensure that a loop has a single loop counter, an optional control variable, and is not degenerate.	2 - Small differences	A6-5-2	
6.2.3 Do not alter a control or counter variable more than once in a loop.	3 - Significant differences	M6-5-3	It is prohibited to alter a control or counter variable within condition or statement of a loop.
6.2.4 Only modify a for loop counter in the for expression.	2 - Small differences	M6-5-3	
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.	2 - Small differences	M6-6-1	
6.3.2 Ensure that execution of a function with a non-void return type ends in a return statement with a value.	2 - Small differences	A8-4-2	
6.4.1 Postpone variable definitions as long as possible.	2 - Small differences	M3-4-1	
7.1.1 Declare each identifier on a separate line in a separate declaration.	2 - Small differences	A7-1-7	
7.1.2 Use const whenever possible.	2 - Small differences	A7-1-1, A7-1-2	
7.1.3 Do not place type specifiers before non-type specifiers in a declaration.	2 - Small differences	A7-1-8	
7.1.4 Place CV-qualifiers on the right hand side of the type they apply to.	3 - Significant differences	A7-1-3	Placement of cv-qualifiers is only restricted for typedefs.
7.1.5 Do not inline large functions.	4 - Rejected		Code metrics are not covered by AUTOSAR C++ Coding Guidelines.
7.1.6 Use class types or typedefs to abstract scalar quantities and standard integer types.	3 - Significant differences	A3-9-1	AUTOSAR C++ Coding Guidelines forces to use typedefs for built-in numerical types.
7.1.7 Use a trailing return type in preference to type disambiguation using typename.	2 - Small differences	A8-2-1	

7.1.8 Use auto id = expr when declaring a variable to have the same type as its initializer function call.	3 - Significant differences	A7-1-5	The rule is formulated differently.
7.1.9 Do not explicitly specify the return type of a lambda.	4 - Rejected		To avoid implicit type conversion return type of lambda expression needs to be specified explicitly, see: A5-1-6.
7.1.10 Use static_assert for assertions involving compile time constants.	3 - Significant differences	A16-6-1	It is recommended to use the static_assert instead of #error directive.
7.2.1 Use an explicit enumeration base and ensure that it is large enough to store all enumerators.	2 - Small differences	A7-2-2	
7.2.2 Initialize none, the first only or all enumerators in an enumeration.	2 - Small differences	A7-2-4	
7.3.1 Do not use using directives.	2 - Small differences	M7-3-4	
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.			
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.	2 - Small differences	A3-1-1, M3-2-2	
7.4.3 Ensure that an object or a function used from multiple translation units is declared in a single header file.	2 - Small differences	A3-1-1, M3-2-4	
7.5.1 Do not use the asm declaration.	2 - Small differences	A7-4-1	
8.1.1 Do not use multiple levels of pointer indirection.	3 - Significant differences	A5-0-3	At most two levels of pointer indirection are allowed.
8.2.1 Make parameter names absent or identical in all declarations.	2 - Small differences	M3-9-1	
8.2.2 Do not declare functions with an excessive number of parameters.	4 - Rejected		Code metrics are not covered by AUTOSAR C++ Coding Guidelines.
8.2.3 Pass small objects with a trivial copy constructor by value.	4 - Rejected		The rule is vague, “small” has no technical meaning.

8.2.4 Do not pass std::unique_ptr by const reference.	3 - Significant differences	A8-4-11, A8-4-12	A8-4-12 covers how to pass a `std::unique_ptr`, A8-4-11 covers when not to pass by `std::unique_ptr`.
8.3.1 Do not write functions with an excessive McCabe Cyclomatic Complexity.	4 - Rejected		Code metrics are not covered by AUTOSAR C++ Coding Guidelines.
8.3.2 Do not write functions with a high static program path count.	4 - Rejected		Code metrics are not covered by AUTOSAR C++ Coding Guidelines.
8.3.3 Do not use default arguments.	4 - Rejected		Using default arguments is allowed with some restrictions, see e.g. M8-3-1.
8.3.4 Define =delete functions with parameters of type rvalue reference to const.	2 - Small differences	A13-3-1, A18-9-3	
8.4.1 Do not access an invalid object or an object with indeterminate value.	2 - Small differences	A8-5-0, A12-8-3	
8.4.2 Ensure that a braced aggregate initializer matches the layout of the aggregate object.	2 - Small differences	M8-5-2	
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.	2 - Small differences	M9-3-3	
9.1.2 Make default arguments the same or absent when overriding a virtual function.	2 - Small differences	M8-3-1	
9.1.3 Do not return non-const handles to class data from const member functions.	2 - Small differences	M9-3-1, A9-3-1	
9.1.4 Do not write member functions which return non-const handles to data less accessible than the member function.	3 - Significant differences	A9-3-1	It is allowed to return non-const handles to static data.
9.1.5 Do not introduce virtual functions in a final class.	2 - Small differences	A10-3-3	

9.2.1 Declare bit-fields with an explicitly unsigned integral or enumeration type.	3 - Significant differences	A9-6-1	Any type with a defined size is allowed to be used for a bit-field.
10.1.1 Ensure that access to base class subobjects does not require explicit disambiguation.	3 - Significant differences	A10-1-1	Inheritance from more than one base class is prohibited.
10.2.1 Use the override special identifier when overriding a virtual function.	2 - Small differences	A10-3-2	
10.3.1 Ensure that a derived class has at most one base class which is not an interface class.	2 - Small differences	A10-1-1	Note that the definition of an interface changed, see: `Interface-Class`.
11.1.1 Declare all data members private.	2 - Small differences	M11-0-1	
11.2.1 Do not use friend declarations.	2 - Small differences	A11-3-1	
12.1.1 Do not declare implicit user defined conversions.	3 - Significant differences	A12-1-4, A13-5-2	
12.2.1 Declare virtual, private or protected the destructor of a type used as a base class.	3 - Significant differences	A12-4-1	Destructor of a base class shall be public virtual, public override or protected non-virtual.
12.3.1 Correctly declare overloads for operator new and delete.	2 - Small differences	A18-5-11	
12.4.1 Do not use the dynamic type of an object unless the object is fully constructed.	2 - Small differences	M12-1-1	
12.4.2 Ensure that a constructor initializes explicitly all base classes and non-static data members.	2 - Small differences	A12-1-1	
12.4.3 Do not specify both an NSDMI and a member initializer in a constructor for the same non static member.	2 - Significant differences	A12-1-2	Using both NSDMI and member initializer list in one class is not allowed.
12.4.4 Write members in an initialization list in the order in which they are declared.	2 - Small differences	A8-5-1	
12.4.5 Use delegating constructors to reduce code duplication.	2 - Small differences	A12-1-5	

12.5.1 Define explicitly =default or =delete implicit special member functions of concrete classes.	3 - Significant differences	A12-0-1	
12.5.2 Define special members =default if the behavior is equivalent.	2 - Small differences	A12-7-1	
12.5.3 Ensure that a user defined move/copy constructor only moves/copies base and member objects.	2 - Small differences	A12-8-1	
12.5.4 Declare noexcept the move constructor and move assignment operator.	3 - Significant differences	A15-5-1	AUTOSAR C++ Coding Guidelines requires additional functions to be noexcept.
12.5.5 Correctly reset moved-from handles to resources in the move constructor.	2 - Small differences	A12-8-1	
12.5.6 Use an atomic, non-throwing swap operation to implement the copy and move assignment operators.	2 - Small differences	A12-8-2	
12.5.7 Declare assignment operators with the ref-qualifier &.	2 - Small differences	A12-8-7	
12.5.8 Make the copy assignment operator of an abstract class protected or define it =delete.	3 - Significant differences	A12-8-6	AUTOSAR C++ Coding Guidelines requires additional functions to be comply with this rule.
13.1.1 Ensure that all overloads of a function are visible from where it is called.	2 - Small differences	A7-3-1	
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.	3 - Significant differences	A13-3-1	A function taking “forwarding reference” shall not be overloaded.
13.2.1 Do not overload operators with special semantics.	2 - Small differences	M5-2-11, M5-3-3	
13.2.2 Ensure that the return type of an overloaded binary operator matches the built-in counterparts.	2 - Small differences	A13-2-1, A13-2-2, A13-2-3	
13.2.3 Declare binary arithmetic and bitwise operators as non-members.	4 - Rejected		Non-generic design principle; There is no need for a new rule.
13.2.4 When overloading the subscript operator (operator[]) implement both const and non-const versions.	2 - Small differences	A13-5-1	

13.2.5 Implement a minimal set of operators and use them to implement all other related operators.	4 - Rejected		Non-generic design principle; There is no need for a new rule.
14.1.1 Use variadic templates rather than an ellipsis.	3 - Significant differences	A8-4-1	AUTOSAR C++ Coding Guidelines prohibits usage of variadic arguments.
14.2.1 Declare template specializations in the same file as the primary template they specialize.	3 - Significant differences	A14-7-2	Allowed to declare in a header file that declares user-defined type, for which the specialization is declared.
14.2.2 Do not explicitly specialize a function template that is overloaded with other templates.	3 - Significant differences	A14-8-2	Function templates specialization is forbidden.
14.2.3 Declare extern an explicitly instantiated template.	4 - Rejected		
15.1.1 Only use instances of std::exception for exceptions.	2 - Small differences	A15-1-1	
15.2.1 Do not throw an exception from a destructor.	2 - Small differences	A15-5-1	
15.3.1 Do not access non-static members from a catch handler of constructor/destructor function try block.	2 - Small differences	M15-3-3	
15.3.2 Ensure that a program does not result in a call to std::terminate.	2 - Small differences	A15-5-2, A15-5-3	
16.1.1 Use the preprocessor only for implementing include guards, and including header files with include guards.	3 - Significant differences	A16-0-1	Conditional and unconditional file inclusion is allowed.
16.1.2 Do not include a path specifier in filenames supplied in #include directives.	3 - Significant differences	A16-2-1	Path specifier /is allowed to specify a path relative to path passed to the compiler.
16.1.3 Match the filename in a #include directive to the one on the file system.	4 - Rejected		

16.1.4 Use <> brackets for system and standard library headers. Use quotes for all other headers.	4 - Rejected		The rule defines a coding style. Anyway, these are the only forms allowed by the C++ Language Standard. No need for a new rule.
16.1.5 Include directly the minimum number of headers required for compilation.	4 - Rejected		There shall be no unused include directives, however all needed headers shall be included explicitly. See: A16-2-2, A16-2-3.
17.1.1 Do not use std::vector<bool>.	2 - Small differences	A18-1-2	
17.2.1 Wrap use of the C Standard Library.	2 - Small differences	A17-1-1	
17.3.1 Do not use std::move on objects declared with const or const & type.	2 - Small differences	A18-9-3	
17.3.2 Use std::forward to forward universal references.	2 - Small differences	A18-9-2	
17.3.3 Do not subsequently use the argument to std::forward.	2 - Small differences	A18-9-4	
17.3.4 Do not create smart pointers of array type.	3 - Significant differences	A18-1-4	Rule reformulated to better capture the problem cases and to allow use of smart pointer specializations for array types.
17.3.5 Do not create an rvalue reference of std::array.	4 - Rejected		The rule is only a hint saying that passing std::array by rvalue reference would be less efficient than passing it by reference. However, usage depends on the case, and it should be allowed to pass std::array by rvalue reference.
17.4.1 Use const container calls when result is immediately converted to a const iterator.	2 - Small differences	A23-0-1	

17.4.2 Use API calls that construct objects in place.	3 - Significant differences	A18-5-2	A18-5-2 prohibits explicit calls to new and delete operators, std::make_shared, std::make_unique and similar constructions are recommended.
17.5.1 Do not ignore the result of std::remove, std::remove_if or std::unique.	2 - Small differences	A0-1-2	
18.1.1 Do not use platform specific multi-threading facilities.	5 - Not yet analyzed		The “Concurrency” chapter is not yet covered, this will be addressed in future.
18.2.1 Use high_integrity::thread in place of std::thread.	4 - Rejected		The high_integrity::thread is not part of the C++ Language Standard.
18.2.2 Synchronize access to data shared between threads using a single lock.	5 - Not yet analyzed		The “Concurrency” chapter is not yet covered, this will be addressed in future.
18.2.3 Do not share volatile data between threads.	3 - Significant differences	A2-11-1	Volatile keyword forbidden.
18.2.4 Use std::call_once rather than the Double-Checked Locking pattern.	5 - Not yet analyzed		The “Concurrency” chapter is not yet covered, this will be addressed in future.
18.3.1 Within the scope of a lock, ensure that no static path results in a lock of the same mutex.	5 - Not yet analyzed		The “Concurrency” chapter is not yet covered, this will be addressed in future.
18.3.2 Ensure that order of nesting of locks in a project forms a DAG.	5 - Not yet analyzed		The “Concurrency” chapter is not yet covered, this will be addressed in future.

18.3.3 Do not use std::recursive_mutex.	5 - Not yet analyzed		The “Concurrency” chapter is not yet covered, this will be addressed in future.
18.3.4 Only use std::unique_lock when std::lock_guard cannot be used.	5 - Not yet analyzed		The “Concurrency” chapter is not yet covered, this will be addressed in future.
18.3.5 Do not access the members of std::mutex directly.	5 - Not yet analyzed		The “Concurrency” chapter is not yet covered, this will be addressed in future.
18.3.6 Do not use relaxed atomics.	5 - Not yet analyzed		The “Concurrency” chapter is not yet covered, this will be addressed in future.
18.4.1 Do not use std::condition_variable_any on a std::mutex	5 - Not yet analyzed		The “Concurrency” chapter is not yet covered, this will be addressed in future.

Table B.2: HIC++ v4.0

B.3 Traceability to JSF

The following table demonstrates the traceability to Joint Strike Fighter Air Vehicle C++ Coding Standard [8]. This is not considered as a reproduction, but a mean to compare the two standards.

Note that the copyright of JSF-AV 2005 allows an unlimited distribution anyway.

JSF Rule:	Relation type:	Related rule:	Comment:
AV Rule 8 All code shall conform to ISO/IEC 14882:2002(E) standard C++.	2 - Small differences	A1-1-1	
AV Rule 9 Only those characters specified in the C++ basic source character set will be used. [...].	2 - Small differences	A2-3-1	

AV Rule 10 Values of character types will be restricted to a defined and documented subset of ISO 10646-1.	4 - Rejected		Source files encoding is too restrictive and not covered by AUTOSAR C++ Coding Guidelines.
AV Rule 11 Trigraphs will not be used.	2 - Small differences	A2-5-1	
AV Rule 12 The following digraphs will not be used [...].	2 - Small differences	A2-5-2	
AV Rule 13 Multi-byte characters and wide string literals will not be used.	4 - Rejected		Agreed for wchar_t type only, A2-13-3.
AV Rule 14 Literal suffixes shall use uppercase rather than lowercase letters.	2 - Small differences	M2-13-4	
AV Rule 15 Provision shall be made for run-time checking (defensive programming).	2 - Small differences	M0-3-1	
AV Rule 16 Only DO-178B level A [15] certifiable or SEAL 1 C/C++ libraries shall be used with safety-critical (i.e. SEAL 1) code.	4 - Rejected		JSF-specific rule.
AV Rule 17 The error indicator errno shall not be used.	2 - Small differences	M19-3-1	
AV Rule 18 The macro offsetof, in library <stddef.h>, shall not be used.	2 - Small differences	M18-2-1	
AV Rule 19 <locale.h> and the setlocale function shall not be used.	2 - Small differences	A18-0-3	
AV Rule 20 The setjmp macro and the longjmp function shall not be used.	2 - Small differences	M17-0-5	
AV Rule 21 The signal handling facilities of <signal.h> shall not be used.	2 - Small differences	M18-7-1	
AV Rule 22 The input /output library <stdio.h> shall not be used.	2 - Small differences	M27-0-1	
AV Rule 23 The library functions atof, atoi and atol from library <stdlib.h> shall not be used.	2 - Small differences	A18-0-2	
AV Rule 24 The library functions abort, exit, getenv and system from library <stdlib.h> shall not be used.	2 - Small differences	M18-0-3	
AV Rule 25 The time handling functions of library <time.h> shall not be used.	2 - Small differences	M18-0-4	
AV Rule 26 Only the following pre-processor directives shall be used: 1. #ifndef 2. #define 3. #endif 4. #include.	2 - Small differences	A16-0-1	

AV Rule 27 #ifndef, #define and #endif will be used to prevent multiple inclusions of the same header file. Other techniques to prevent the multiple inclusions of header files will not be used.	2 - Small differences	A16-0-1, M16-2-3	
AV Rule 28 The #ifndef and #endif pre-processor directives will only be used as defined in AV Rule 27 to prevent multiple inclusions of the same header file.	2 - Small differences	A16-0-1	
AV Rule 29 The #define pre-processor directive shall not be used to create inline macros. Inline functions shall be used instead.	2 - Small differences	A16-0-1	
AV Rule 30 The #define pre-processor directive shall not be used to define constant values. Instead, the const qualifier shall be applied to variable declarations to specify constant values.	2 - Small differences	A16-0-1	
AV Rule 31 The #define pre-processor directive will only be used as part of the technique to prevent multiple inclusions of the same header file.	2 - Small differences	A16-0-1	
AV Rule 32 The #include pre-processor directive will only be used to include header (*.h) files.	2 - Small differences	A16-0-1	
AV Rule 33 The #include directive shall use the <filename.h> notation to include header files.	4 - Rejected		Including files using quotes is also possible.
AV Rule 34 Header files should contain logically related declarations only.	2 - Small differences	A3-3-1	
AV Rule 35 A header file will contain a mechanism that prevents multiple inclusions of itself.	2 - Small differences	M16-2-3	
AV Rule 36 Compilation dependencies should be minimized when possible.	4 - Rejected		The rule is vague; more precisely explained by AV Rules 37 and 38.
AV Rule 37 Header (include) files should include only those header files that are required for them to successfully compile. Files that are only used by the associated .cpp file should be placed in the .cpp file - not the .h file.	2 - Small differences	A16-2-2, A16-2-3	

AV Rule 38 Declarations of classes that are only accessed via pointers (*) or references (&) should be supplied by forward headers that contain only forward declarations.	3 - Significant differences	A16-2-3	Forward declarations considered as a possible solution for unnecessarry inclusions.
AV Rule 39 Header files (*.h) will not contain non-const variable definitions or function definitions.	2 - Small differences	M3-2-4, A3-3-1	
AV Rule 40 Every implementation file shall include the header files that uniquely define the inline functions, types, and templates used.	2 - Small differences	M3-2-4, A3-3-1	
AV Rule 41 Source lines will be kept to a length of 120 characters or less.	4 - Rejected		Coding style is not covered by AUTOSAR C++ Coding Guidelines.
AV Rule 42 Each expression-statement will be on a separate line.	2 - Small differences	A7-1-7	
AV Rule 43 Tabs should be avoided.	4 - Rejected		Coding style is not covered by AUTOSAR C++ Coding Guidelines.
AV Rule 44 All indentations will be at least two spaces and be consistent within the same source file.	4 - Rejected		Coding style is not covered by AUTOSAR C++ Coding Guidelines.
AV Rule 45 All words in an identifier will be separated by the “_” character.	4 - Rejected		Coding style is not covered by AUTOSAR C++ Coding Guidelines.
AV Rule 46 User-specified identifiers (internal and external) will not rely on significance of more than 64 characters.	4 - Rejected		Coding style is not covered by AUTOSAR C++ Coding Guidelines.
AV Rule 47 Identifiers will not begin with the underscore character “_”.	3 - Significant differences	A17-0-1	

AV Rule 48 Identifiers will not differ by: (a) Only a mixture of case, (b) The presence/absence of the underscore character, (c) The interchange of the letter “O”, with the number “0” or the letter “D”, (d) The interchange of the letter “I”, with the number “1” or the letter “l”, (e) The interchange of the letter “S” with the number “5”, (f) The interchange of the letter “Z” with the number “2”, (g) The interchange of the letter “n” with the letter “h”.	4 - Rejected		Coding style is not covered by AUTOSAR C++ Coding Guidelines.
AV Rule 49 All acronyms in an identifier will be composed of uppercase letters.	4 - Rejected		Coding style is not covered by AUTOSAR C++ Coding Guidelines.
AV Rule 50 The first word of the name of a class, structure, namespace, enumeration, or type created with typedef will begin with an uppercase letter. All others letters will be lowercase.	4 - Rejected		Coding style is not covered by AUTOSAR C++ Coding Guidelines.
AV Rule 51 All letters contained in function and variable names will be composed entirely of lowercase letters.	4 - Rejected		Coding style is not covered by AUTOSAR C++ Coding Guidelines.
AV Rule 52 Identifiers for constant and enumerator values shall be lowercase.	4 - Rejected		Coding style is not covered by AUTOSAR C++ Coding Guidelines.
AV Rule 53 Header files will always have a file name extension of “.h”.	3 - Significant differences	A3-1-2	
AV Rule 53.1 The following character sequences shall not appear in header file names: ’, \, /*, //, or ”.	2 - Small differences	A16-2-1	
AV Rule 54 Implementation files will always have a file name extension of “.cpp”.	2 - Small differences	A3-1-3	
AV Rule 55 The name of a header file should reflect the logical entity for which it provides declarations.	2 - Small differences	A2-8-1	
AV Rule 56 The name of an implementation file should reflect the logical entity for which it provides definitions and have a “.cpp” extension (this name will normally be identical to the header file that provides the corresponding declarations.)	2 - Small differences	A2-8-2	

AV Rule 57 The public, protected, and private sections of a class will be declared in that order (the public section is declared before the protected section which is declared before the private section).	4 - Rejected		Coding style is not covered by AUTOSAR C++ Coding Guidelines.
AV Rule 58 When declaring and defining functions with more than two parameters, the leading parenthesis and the first argument will be written on the same line as the function name. Each additional argument will be written on a separate line (with the closing parenthesis directly after the last argument).	4 - Rejected		Coding style is not covered by AUTOSAR C++ Coding Guidelines.
AV Rule 59 The statements forming the body of an if, else if, else, while, do...while or for statement shall always be enclosed in braces, even if the braces form an empty block.	2 - Small differences	M6-3-1	
AV Rule 60 Braces (“{}”) which enclose a block will be placed in the same column, on separate lines directly before and after the block.	4 - Rejected		Coding style is not covered by AUTOSAR C++ Coding Guidelines.
AV Rule 61 Braces (“{}”) which enclose a block will have nothing else on the line except comments (if necessary).	4 - Rejected		Coding style is not covered by AUTOSAR C++ Coding Guidelines.
AV Rule 62 The dereference operator “*” and the address-of operator “&” will be directly connected with the type-specifier.	4 - Rejected		Coding style is not covered by AUTOSAR C++ Coding Guidelines.
AV Rule 63 Spaces will not be used around “.” or “->”, nor between unary operators and operands.	4 - Rejected		Coding style is not covered by AUTOSAR C++ Coding Guidelines.
AV Rule 64 A class interface should be complete and minimal.	4 - Rejected		Code metrics are not covered by AUTOSAR C++ Coding Guidelines.
AV Rule 65 A structure should be used to model an entity that does not require an invariant.	3 - Significant differences	A11-0-2	
AV Rule 66 A class should be used to model an entity that maintains an invariant.	3 - Significant differences	A11-0-1	
AV Rule 67 Public and protected data should only be used in structs - not classes.	2 - Small differences	M11-0-1	

AV Rule 68 Unneeded implicitly generated member functions shall be explicitly disallowed.	3 - Significant differences	A12-0-1	It is allowed to follow both “Rule of zero” and “Rule of five”.
AV Rule 69 A member function that does not affect the state of an object (its instance variables) will be declared const.	2 - Small differences	M9-3-3	
AV Rule 70 A class will have friends only when a function or object requires access to the private elements of the class, but is unable to be a member of the class for logical or efficiency reasons.	4 - Rejected		Friend declarations are prohibited, see: A11-3-1.
AV Rule 70.1 An object shall not be improperly used before its lifetime begins or after its lifetime ends.	2 - Small differences	A3-8-1, A5-1-4, M7-5-1, M7-5-2, A7-5-1, M12-1-1	
AV Rule 71 Calls to an externally visible operation of an object, other than its constructors, shall not be allowed until the object has been fully initialized.	2 - Small differences	A12-1-1, M12-1-1	
AV Rule 71.1 A class’s virtual functions shall not be invoked from its destructor or any of its constructors.	2 - Small differences	M12-1-1	
AV Rule 72 The invariant for a class should be: (a) a part of the postcondition of every class constructor, (b) a part of the precondition of the class destructor (if any), (c) a part of the precondition and postcondition of every other publicly accessible operation.	3 - Significant differences	A15-2-2, M11-0-1, M9-3-1, A9-3-1	No enforcement on pre/postcondition checking in the Coding Guidelines
AV Rule 73 Unnecessary default constructors shall not be defined.	4 - Rejected		No rule needed.
AV Rule 74 Initialization of nonstatic class members will be performed through the member initialization list rather than through assignment in the body of a constructor.	2 - Small differences	A12-6-1	
AV Rule 75 Members of the initialization list shall be listed in the order in which they are declared in the class.	2 - Small differences	A8-5-1	
AV Rule 76 A copy constructor and an assignment operator shall be declared for classes that contain pointers to data items or nontrivial destructors.	2 - Small differences	A12-7-1	

AV Rule 77 A copy constructor shall copy all data members and bases that affect the class invariant (a data element representing a cache, for example, would not need to be copied).	2 - Small differences	A12-8-1	
AV Rule 77.1 The definition of a member function shall not contain default arguments that produce a signature identical to that of the implicitly-declared copy constructor for the corresponding class/structure.	4 - Rejected		No ambiguity in C++14, see ISO/IEC 14882:2014 C++14 12.8 [class.copy]2
AV Rule 78 All base classes with a virtual function shall define a virtual destructor.	2 - Small differences	A12-4-1	
AV Rule 79 All resources acquired by a class shall be released by the class's destructor.	2 - Small differences	A15-1-4, A18-5-2	
AV Rule 80 The default copy and assignment operators will be used for classes when those operators offer reasonable semantics.	2 - Small differences	A12-0-1	
AV Rule 81 The assignment operator shall handle self-assignment correctly.	2 - Small differences	A12-8-5	
AV Rule 82 An assignment operator shall return a reference to *this.	2 - Small differences	A13-2-1	
AV Rule 83 An assignment operator shall assign all data members and bases that affect the class invariant (a data element representing a cache, for example, would not need to be copied).	2 - Small differences	A6-2-1	
AV Rule 84 Operator overloading will be used sparingly and in a conventional manner.	4 - Rejected		The rule is vague. Design principle; There is no need for a new rule.
AV Rule 85 When two operators are opposites (such as == and !=), both will be defined and one will be defined in terms of the other.	2 - Small differences	A13-5-4	
AV Rule 86 Concrete types should be used to represent simple independent concepts.	4 - Rejected		The rule is vague. Design principle; There is no need for a new rule.
AV Rule 87 Hierarchies should be based on abstract classes.	2 - Small differences	A10-4-1	
AV Rule 88 Multiple inheritance shall only be allowed in the following restricted form: n interfaces plus m private implementations, plus at most one protected implementation.	3 - Significant differences	A10-1-1	

AV Rule 88.1 A stateful virtual base shall be explicitly declared in each derived class that accesses it.	4 - Rejected		Virtual inheritance should not be used, see: M10-1-1.
AV Rule 89 A base class shall not be both virtual and non-virtual in the same hierarchy.	2 - Small differences	M10-1-3	
AV Rule 90 Heavily used interfaces should be minimal, general and abstract.	4 - Rejected		The rule is vague. Design principle; There is no need for a new rule.
AV Rule 91 Public inheritance will be used to implement “is-a” relationships.	2 - Small differences	A10-0-1	
AV Rule 92 A subtype (publicly derived classes) will conform to the following guidelines with respect to all classes involved in the polymorphic assignment of different subclass instances to the same variable or parameter during the execution of the system: (1) Preconditions of derived methods must be at least as weak as the preconditions of the methods they override. (2) Postconditions of derived methods must be at least as strong as the postconditions of the methods they override. In other words, subclass methods must expect less and deliver more than the base class methods they override. This rule implies that subtypes will conform to the Liskov Substitution Principle.	4 - Rejected		The rule is vague. Design principle; There is no need for a new rule.
AV Rule 93 “has-a” or “is-implemented-in-terms-of” relationships will be modeled through membership or non-public inheritance.	2 - Small differences	A10-0-2	
AV Rule 94 An inherited nonvirtual function shall not be redefined in a derived class.	2 - Small differences	A10-2-1	
AV Rule 95 An inherited default parameter shall never be redefined.	2 - Small differences	M8-3-1	
AV Rule 96 Arrays shall not be treated polymorphically.	2 - Small differences	A5-0-4	
AV Rule 97 Arrays shall not be used in interfaces. Instead, the Array class should be used.	2 - Small differences	M5-2-12	
AV Rule 97.1 Neither operand of an equality operator (== or !=) shall be a pointer to a virtual member function.	3 - Significant differences	A5-10-1	Testing for equality with null-pointer-constant is allowed.

AV Rule 98 Every nonlocal name, except main(), should be placed in some namespace.	2 - Small differences	M7-3-1	
AV Rule 99 Namespaces will not be nested more than two levels deep.	4 - Rejected		Not compliant with the AUTOSAR general requirements.
AV Rule 100 Elements from a namespace should be selected as follows: (a) using declaration or explicit qualification for few (approximately five) names, (b) using directive for many names.	3 - Significant differences	M7-3-4, M7-3-6	
AV Rule 101 Templates shall be reviewed as follows: (1) with respect to the template in isolation considering assumptions or requirements placed on its arguments, (2) with respect to all functions instantiated by actual arguments.	4 - Rejected		Rule concerns code review process.
AV Rule 102 Template tests shall be created to cover all actual template instantiations.	4 - Rejected		Rule concerns test process.
AV Rule 103 Constraint checks should be applied to template arguments.	2 - Small differences	A14-1-1	
AV Rule 104 A template specialization shall be declared before its use.	2 - Small differences	A14-7-2	
AV Rule 105 A template definition's dependence on its instantiation contexts should be minimized.	4 - Rejected		This rule is vague. "Minimized" has no technical meaning.
AV Rule 106 Specializations for pointer types should be made where appropriate.	4 - Rejected		This rule is vague. "Where appropriate" has no technical meaning.
AV Rule 107 Functions shall always be declared at file scope.	2 - Small differences	M3-1-2	
AV Rule 108 Functions with variable numbers of arguments shall not be used.	2 - Small differences	A8-4-1	
AV Rule 109 A function definition should not be placed in a class specification unless the function is intended to be inlined.	3 - Significant differences	A3-1-5	Also included template methods and methods of template classes.
AV Rule 110 Functions with more than 7 arguments will not be used.	4 - Rejected		Code metrics are not covered by AUTOSAR C++ Coding Guidelines.

AV Rule 111 A function shall not return a pointer or reference to a non-static local object.	2 - Small differences	M7-5-2	
AV Rule 112 Function return values should not obscure resource ownership.	2 - Small differences	M9-3-1, A9-3-1, A20-8-2, A20-8-3	
AV Rule 113 Functions will have a single exit point.	4 - Rejected		See Single-point-of-exit.
AV Rule 114 All exit points of value-returning functions shall be through return statements.	2 - Small differences	A8-4-2	
AV Rule 115 If a function returns error information, then that error information will be tested.	2 - Small differences	A8-4-2	
AV Rule 116 Small, concrete-type arguments (two or three words in size) should be passed by value if changes made to formal parameters should not be reflected in the calling function.	2 - Small differences	A8-4-7	
AV Rule 117 Arguments should be passed by reference if NULL values are not possible.	2 - Small differences	A8-4-10	
AV Rule 117.1 An object should be passed as const T& if the function should not change the value of the object.	2 - Small differences	A8-4-7	
AV Rule 117.2 An object should be passed as T& if the function may change the value of the object.	2 - Small differences	A8-4-9	
AV Rule 118 Arguments should be passed via pointers if NULL values are possible.	3 - Significant differences	A8-4-10	
AV Rule 118.1 An object should be passed as const T* if its value should not be modified.	3 - Significant differences	A8-4-10	
AV Rule 118.2 An object should be passed as T* if its value may be modified.	3 - Significant differences	A8-4-10	
AV Rule 119 Functions shall not call themselves, either directly or indirectly (i.e. recursion shall not be allowed).	2 - Small differences	A7-5-2	
AV Rule 120 Overloaded operations or methods should form families that use the same semantics, share the same name, have the same purpose, and that are differentiated by formal parameters.	4 - Rejected		This rule is vague. It is not a coding rule.
AV Rule 121 Only functions with 1 or 2 statements should be considered candidates for inline functions.	4 - Rejected		Code metrics are not covered by AUTOSAR C++ Coding Guidelines.

AV Rule 122 Trivial accessor and mutator functions should be inlined.	2 - Small differences	A3-1-6	
AV Rule 123 The number of accessor and mutator functions should be minimized.	4 - Rejected		This rule is vague. It is not a coding rule.
AV Rule 124 Trivial forwarding functions should be inlined.	4 - Rejected		Forwarding functions are mostly templates and they most likely will be inlined.
AV Rule 125 Unnecessary temporary objects should be avoided.	2 - Small differences	M0-1-9, A1-4-3	
AV Rule 126 Only valid C++ style comments (//) shall be used.	2 - Small differences	M2-7-1	
AV Rule 127 Code that is not used (commented out) shall be deleted.	2 - Small differences	A2-7-2	
AV Rule 128 Comments that document actions or sources (e.g. tables, figures, paragraphs, etc.) outside of the file being documented will not be allowed.	2 - Small differences	A2-7-5	
AV Rule 129 Comments in header files should describe the externally visible behavior of the functions or classes being documented.	3 - Significant differences	A2-7-3	
AV Rule 130 The purpose of every line of executable code should be explained by a comment, although one comment may describe more than one line of code.	4 - Rejected		
AV Rule 131 One should avoid stating in comments what is better stated in code (i.e. do not simply repeat what is in the code).	4 - Rejected		
AV Rule 132 Each variable declaration, typedef, enumeration value, and structure member will be commented.	4 - Rejected		
AV Rule 133 Every source file will be documented with an introductory comment that provides information on the file name, its contents, and any program-required information (e.g. legal statements, copyright information, etc).	4 - Rejected		
AV Rule 134 Assumptions (limitations) made by functions should be documented in the function's preamble.	3 - Significant differences	A2-7-3	
AV Rule 135 Identifiers in an inner scope shall not use the same name as an identifier in an outer scope, and therefore hide that identifier.	2 - Small differences	A2-10-1	

AV Rule 136 Declarations should be at the smallest feasible scope.	2 - Small differences	M3-4-1	
AV Rule 137 All declarations at file scope should be static where possible.	3 - Significant differences	A3-3-1	
AV Rule 138 Identifiers shall not simultaneously have both internal and external linkage in the same translation unit.	2 - Small differences	M3-3-2	
AV Rule 139 External objects will not be declared in more than one file.	2 - Small differences	M3-2-3	
AV Rule 140 The register storage class specifier shall not be used.	2 - Small differences	A7-1-4	
AV Rule 141 A class, structure, or enumeration will not be declared in the definition of its type.	2 - Small differences	A7-1-9	
AV Rule 142 All variables shall be initialized before use.	2 - Small differences	A8-5-0	
AV Rule 143 Variables will not be introduced until they can be initialized with meaningful values.	2 - Small differences	M3-4-1	
AV Rule 144 Braces shall be used to indicate and match the structure in the non-zero initialization of arrays and structures.	2 - Small differences	M8-5-2	
AV Rule 145 In an enumerator list, the “=” construct shall not be used to explicitly initialize members other than the first, unless all items are explicitly initialized.	2 - Small differences	A7-2-4	
AV Rule 146 Floating point implementations shall comply with a defined floating point standard. The standard that will be used is the ANSI/IEEE Std 754	2 - Small differences	A0-4-1	
AV Rule 147 The underlying bit representations of floating point numbers shall not be used in any way by the programmer.	2 - Small differences	M3-9-3	
AV Rule 148 Enumeration types shall be used instead of integer types (and constants) to select from a limited series of choices.	2 - Small differences	A7-2-5	
AV Rule 149 Octal constants (other than zero) shall not be used.	2 - Small differences	M2-13-2	
AV Rule 150 Hexadecimal constants will be represented using all uppercase letters.	2 - Small differences	A2-13-5	
AV Rule 151 Numeric values in code will not be used; symbolic values will be used instead.	2 - Small differences	A5-1-1	
AV Rule 151.1 A string literal shall not be modified.	2 - Small differences	A2-13-4	

AV Rule 152 Multiple variable declarations shall not be allowed on the same line.	2 - Small differences	A7-1-7	
AV Rule 153 Unions shall not be used.	2 - Small differences	A9-5-1	
AV Rule 154 Bit-fields shall have explicitly unsigned integral or enumeration types only.	3 - Significant differences	A9-6-1	Any type with a defined size is allowed to be used for a bit-field.
AV Rule 155 Bit-fields will not be used to pack data into a word for the sole purpose of saving space.	2 - Small differences	A9-6-2	
AV Rule 156 All the members of a structure (or class) shall be named and shall only be accessed via their names.	2 - Small differences	A12-0-2	
AV Rule 157 The right hand operand of a && or ||operator shall not contain side effects.	2 - Small differences	M5-14-1	
AV Rule 158 The operands of a logical && or ||shall be parenthesized if the operands contain binary operators.	2 - Small differences	A5-2-6	
AV Rule 159 Operators ||, &&, and unary & shall not be overloaded.	2 - Small differences	M5-2-11, M5-3-3	
AV Rule 160 An assignment expression shall be used only as the expression in an expression statement.	2 - Small differences	M6-2-1	
AV Rule 162 Signed and unsigned values shall not be mixed in arithmetic or comparison operations.	2 - Small differences	M5-0-4, M5-0-9	
AV Rule 163 Unsigned arithmetic shall not be used.	2 - Small differences	M5-0-4, M5-0-9	
AV Rule 164 The right hand operand of a shift operator shall lie between zero and one less than the width in bits of the left-hand operand (inclusive).	2 - Small differences	M5-8-1	
AV Rule 164.1 The left-hand operand of a right-shift operator shall not have a negative value.	2 - Small differences	M5-8-1	
AV Rule 165 The unary minus operator shall not be applied to an unsigned expression.	2 - Small differences	M5-3-2	
AV Rule 166 The sizeof operator will not be used on expressions that contain side effects.	2 - Small differences	M5-3-4	
AV Rule 167 The implementation of integer division in the chosen compiler shall be determined, documented and taken into account.	3 - Significant differences	A0-4-2	
AV Rule 168 The comma operator shall not be used.	2 - Small differences	M5-18-1	
AV Rule 169 Pointers to pointers should be avoided when possible.	3 - Significant differences	A5-0-3	

AV Rule 170 More than 2 levels of pointer indirection shall not be used.	3 - Significant differences	A5-0-3	
AV Rule 171 Relational operators shall not be applied to pointer types except where both operands are of the same type and point to: (a) the same object, (b) the same function, (c) members of the same object, or (d) elements of the same array (including one past the end of the same array).	2 - Small differences	M5-0-18	
AV Rule 173 The address of an object with automatic storage shall not be assigned to an object which persists after the object has ceased to exist.	2 - Small differences	M7-5-2	
AV Rule 174 The null pointer shall not be de-referenced.	2 - Small differences	A5-3-2	
AV Rule 175 A pointer shall not be compared to NULL or be assigned NULL; use plain 0 instead.	4 - Rejected		Only nullptr constant shall be used, see: A4-10-1.
AV Rule 176 A typedef will be used to simplify program syntax when declaring function pointers.	4 - Rejected		Implementation principle. There is no need for a new rule.
AV Rule 177 User-defined conversion functions should be avoided.	2 - Small differences	A13-5-3	
AV Rule 178 Down casting (casting from base to derived class) shall only be allowed through one of the following mechanism: (a) Virtual functions that act like dynamic casts (most likely useful in relatively simple cases), (b) Use of the visitor (or similar) pattern (most likely useful in complicated cases)	2 - Small differences	M5-2-2, A5-2-6	
AV Rule 179 A pointer to a virtual base class shall not be converted to a pointer to a derived class.	2 - Small differences	M5-2-3	
AV Rule 180 Implicit conversions that may result in a loss of information shall not be used.	2 - Small differences	A4-7-1	
AV Rule 181 Redundant explicit casts will not be used.	2 - Small differences	M0-1-9	
AV Rule 182 Type casting from any type to or from pointers shall not be used.	3 - Significant differences	M5-2-9	Not fully covered.
AV Rule 183 Every possible measure should be taken to avoid type casting.	4 - Rejected		It is not a coding rule.

AV Rule 184 Floating point numbers shall not be converted to integers unless such a conversion is a specified algorithmic requirement or is necessary for a hardware interface.	2 - Small differences	M5-0-5	
AV Rule 185 C++ style casts (const_cast, reinterpret_cast, and static_cast) shall be used instead of the traditional C-style casts.	2 - Small differences	A5-2-2	
AV Rule 186 There shall be no unreachable code.	2 - Small differences	M0-1-1	
AV Rule 187 All non-null statements shall potentially have a side-effect.	2 - Small differences	M0-1-9	
AV Rule 188 Labels will not be used, except in switch statements.	3 - Significant differences	A6-6-1	
AV Rule 189 The goto statement shall not be used.	2 - Small differences	A6-6-1	
AV Rule 190 The continue statement shall not be used.	4 - Rejected		The continue statement usage is allowed within for-loops, see: M6-6-3.
AV Rule 191 The break statement shall not be used (except to terminate the cases of a switch statement).	2 - Small differences	M6-4-5	
AV Rule 192 All if, else if constructs will contain either a final else clause or a comment indicating why a final else clause is not necessary.	2 - Small differences	M6-4-2	
AV Rule 193 Every non-empty case clause in a switch statement shall be terminated with a break statement.	2 - Small differences	M6-4-3, M6-4-5	
AV Rule 194 All switch statements that do not intend to test for every enumeration value shall contain a final default clause.	2 - Small differences	M6-4-6	
AV Rule 195 A switch expression will not represent a Boolean value.	2 - Small differences	M6-4-7	
AV Rule 196 Every switch statement will have at least two cases and a potential default.	2 - Small differences	A6-4-1	
AV Rule 197 Floating point variables shall not be used as loop counters.	2 - Small differences	M6-5-2	
AV Rule 198 The initialization expression in a for loop will perform no actions other than to initialize the value of a single for loop parameter.	2 - Small differences	A6-5-4	
AV Rule 199 The increment expression in a for loop will perform no action other than to change a single loop parameter to the next value for the loop.	2 - Small differences	A6-5-4	

AV Rule 200 Null initialize or increment expressions in for loops will not be used; a while loop will be used instead.	3 - Significant differences	A6-5-2	
AV Rule 201 Numeric variables being used within a for loop for iteration counting shall not be modified in the body of the loop.	2 - Small differences	M6-5-3	
AV Rule 202 Floating point variables shall not be tested for exact equality or inequality.	2 - Small differences	M6-2-2	
AV Rule 203 Evaluation of expressions shall not lead to overflow/underflow (unless required algorithmically and then should be heavily documented).	2 - Small differences	M5-19-1, A7-1-2	
AV Rule 204 A single operation with side-effects shall only be used in the following contexts: 1. by itself 2. the right-hand side of an assignment 3. a condition 4. the only argument expression with a side-effect in a function call 5. condition of a loop 6. switch condition 7. single part of a chained operation.	3 - Significant differences	M6-2-1	Some of presented examples are not misleading and some of them are already covered.
AV Rule 204.1 The value of an expression shall be the same under any order of evaluation that the standard permits.	2 - Small differences	A5-0-1	
AV Rule 205 The volatile keyword shall not be used unless directly interfacing with hardware.	3 - Significant differences	A2-11-1	Volatile keyword forbidden.
AV Rule 206 Allocation/deallocation from/to the free store (heap) shall not occur after initialization.	2 - Small differences	A18-5-3	
AV Rule 207 Unencapsulated global data will be avoided.	3 - Significant differences	A3-3-2	Restriction to constant-initialized objects only.
AV Rule 208 C++ exceptions shall not be used (i.e. throw, catch and try shall not be used.)	4 - Rejected		C++ exceptions may be used conditionally.
AV Rule 209 The basic types of int, short, long, float and double shall not be used, but specific-length equivalents should be typedef'd accordingly for each compiler, and these type names used in the code.	2 - Small differences	A3-9-1	

AV Rule 210 Algorithms shall not make assumptions concerning how data is represented in memory (e.g. big endian vs. little endian, base class subobject ordering in derived classes, nonstatic data member ordering across access specifiers, etc.)	2 - Small differences	A12-0-2	
AV Rule 210.1 Algorithms shall not make assumptions concerning the order of allocation of nonstatic data members separated by an access specifier.	2 - Small differences	A12-0-2	
AV Rule 211 Algorithms shall not assume that shorts, ints, longs, floats, doubles or long doubles begin at particular addresses.	2 - Small differences	A12-0-2	
AV Rule 212 Underflow or overflow functioning shall not be depended on in any special way.	2 - Small differences	A4-7-1	
AV Rule 213 No dependence shall be placed on C++'s operator precedence rules, below arithmetic operators, in expressions.	2 - Small differences	A5-0-1	
AV Rule 214 Assuming that non-local static objects, in separate translation units, are initialized in a special order shall not be done.	2 - Small differences	A3-3-2	Intent of the rule achieved by restriction to constant-initialized objects.
AV Rule 215 Pointer arithmetic will not be used.	3 - Significant differences	M5-0-15	Pointer arithmetic may be used for array indexing.
AV Rule 216 Programmers should not attempt to prematurely optimize code.	4 - Rejected		It is not a coding rule.
AV Rule 217 Compile-time and link-time errors should be preferred over run-time errors.	4 - Rejected	A16-6-1, A14-1-1, A1-4-3	This rule is too generic, some cases are covered.
AV Rule 218 Compiler warning levels will be set in compliance with project policies.	3 - Significant differences	A1-1-2	
AV Rule 219 All tests applied to a base class interface shall be applied to all derived class interfaces as well. If the derived class poses stronger postconditions/invariants, then the new postconditions /invariants shall be substituted in the derived class tests.	4 - Rejected		It is not a coding rule.
AV Rule 220 Structural coverage algorithms shall be applied against flattened classes.	4 - Rejected		It is not a coding rule.

AV Rule 221 Structural coverage of a class within an inheritance hierarchy containing virtual functions shall include testing every possible resolution for each set of identical polymorphic references.	4 - Rejected		It is not a coding rule.

Table B.3: JSF

B.4 Traceability to SEI CERT C++

The following table demonstrates the traceability to SEI CERT C++ Coding Standard [10]. This is not considered as a reproduction, but a mean to compare the two standards.

Note that the copyright of SEI CERT C++ Coding Standard allows an unlimited distribution anyway.

SEI CERT Rule:	Relation type:	Related rule:	Comment:
DCL30-C. Declare objects with appropriate storage durations.	2 - Small differences	M7-5-2	
DCL40-C. Do not create incompatible declarations of the same function or object.	2 - Small differences	M3-9-1	
DCL50-CPP. Do not define a C-style variadic function.	2 - Small differences	A8-4-1	
DCL51-CPP. Do not declare or define a reserved identifier.	2 - Small differences	A13-1-2, A17-0-1	
DCL52-CPP. Never qualify a reference type with const or volatile.	2 - Small differences	A1-1-1	Covered by a more general rule.
DCL53-CPP. Do not write syntactically ambiguous declarations.	2 - Small differences	A8-5-2	
DCL54-CPP. Overload allocation and deallocation functions as a pair in the same scope.	3 - Significant differences	A18-5-3, A18-5-4	
DCL55-CPP. Avoid information leakage when passing a class object across a trust boundary.	5 - Not yet analyzed		The Security chapter is not yet covered, this will be addressed in future.
DCL56-CPP. Avoid cycles during initialization of static objects.	2 - Small differences	A3-3-2	Intent of the rule achieved by restriction to constant-initialized objects.

DCL57-CPP. Do not let exceptions escape from destructors or deallocation functions.	3 - Significant differences	A15-5-1	AUTOSAR C++ Coding Guidelines specify more functions that need to be noexcept.
DCL58-CPP. Do not modify the standard namespaces.	2 - Small differences	A17-6-1	
DCL59-CPP. Do not define an unnamed namespace in a header file.	2 - Small differences	M7-3-3	
DCL60-CPP. Obey the one-definition rule.	2 - Small differences	M3-2-2	
EXP34-C. Do not dereference null pointers.	2 - Small differences	A5-3-2	
EXP35-C. Do not modify objects with temporary lifetime.	4 - Rejected		Not applicable to C++. See EXP54-CPP.
EXP36-C. Do not cast pointers into more strictly aligned pointer types.	2 - Small differences	A5-2-4, M5-2-8	Direct and indirect casting of pointer types is prohibited via existing rules.
EXP37-C. Call functions with the correct number and type of arguments.	2 - Small differences	M5-2-6	The only case possible in C++ is prohibited via existing rule.
EXP39-C. Do not access a variable through a pointer of an incompatible type.	2 - Small differences	A5-2-2, A5-2-4	Effectively prohibited by existing rules.
EXP42-C. Do not compare padding data.	2 - Small differences	A12-0-2	
EXP45-C. Do not perform assignments in selection statements.	2 - Small differences	A5-0-2, M6-2-1	
EXP46-C. Do not use a bitwise operator with a Boolean-like operand.	4 - Rejected		Use of bitwise operators restricted to following cases: M5-0-10, M5-0-20, M5-0-21.
EXP47-C. Do not call va_arg with an argument of the incorrect type.	4 - Rejected		Use of variable arguments are prohibited, see: A8-4-1.
EXP50-CPP. Do not depend on the order of evaluation for side effects.	2 - Small differences	A5-0-1	

EXP51-CPP. Do not delete an array through a pointer of the incorrect type.	2 - Small differences	M5-2-12, A18-1-1	Covered by a combination of existing rules.
EXP52-CPP. Do not rely on side effects in unevaluated operands.	3 - Significant differences	M5-3-4, A5-3-1	
EXP53-CPP. Do not read uninitialized memory.	2 - Small differences	A8-5-0	
EXP54-CPP. Do not access an object outside of its lifetime.	2 - Small differences	A3-8-1, A5-1-4, M7-5-1, M7-5-2, A7-5-1, M12-1-1	
EXP55-CPP. Do not access a cv-qualified object through a cv-unqualified type.	2 - Small differences	A5-2-3	
EXP56-CPP. Do not call a function with a mismatched language linkage.	2 - Small differences	M5-2-6	
EXP57-CPP. Do not cast or delete pointers to incomplete classes.	3 - Significant differences	A5-3-3, A5-2-2, A5-2-4	The first part of EXP57-CPP is entirely covered by A5-3-3, the second part is implicitly rule out by A5-2-2 and A5-2-4.
EXP58-CPP. Pass an object of the correct type to va_start.	4 - Rejected		Use of variable arguments are prohibited, see: A8-4-1.
EXP59-CPP. Use offsetof() on valid types and members.	4 - Rejected		Use of offsetof() is prohibited, see: M18-2-1.
EXP60-CPP. Do not pass a nonstandard-layout type object across execution boundaries.	2 - Small differences	M1-0-2	
EXP61-CPP. A lambda object must not outlive any of its reference captured objects.	2 - Small differences	A5-1-4	
EXP62-CPP. Do not access the bits of an object representation that are not part of the object's value representation.	2 - Small differences	A12-0-2	
EXP63-CPP. Do not rely on the value of a moved-from object.	2 - Small differences	A12-8-3	
INT30-C. Ensure that unsigned integer operations do not wrap.	2 - Small differences	A4-7-1, M5-19-1	
INT31-C. Ensure that integer conversions do not result in lost or misinterpreted data.	3 - Significant differences	A4-7-1, M5-0-15	
INT32-C. Ensure that operations on signed integers do not result in overflow.	2 - Small differences	A4-7-1	

INT33-C. Ensure that division and remainder operations do not result in divide-by-zero errors.	2 - Small differences	A5-6-1	
INT34-C. Do not shift an expression by a negative number of bits or by greater than or equal to the number of bits that exist in the operand.	2 - Small differences	M5-8-1	
INT35-C. Use correct integer precisions.	3 - Significant differences	A3-9-1	
INT36-C. Converting a pointer to integer or integer to pointer.	2 - Small differences	M5-2-8, M5-2-9	
INT50-CPP. Do not cast to an out-of-range enumeration value.	2 - Small differences	A7-2-1	
CTR50-CPP. Guarantee that container indices and iterators are within the valid range.	2 - Small differences	A5-2-5	
CTR51-CPP. Use valid references, pointers, and iterators to reference elements of a container.	2 - Small differences	A23-0-2	
CTR52-CPP. Guarantee that library functions do not overflow.	2 - Small differences	A5-2-5	
CTR53-CPP. Use valid iterator ranges.	3 - Significant differences	M5-0-16, M5-0-17, A5-2-5	
CTR54-CPP. Do not subtract iterators that do not refer to the same container.	3 - Significant differences	M5-0-16, M5-0-17	
CTR55-CPP. Do not use an additive operator on an iterator if the result would overflow.	3 - Significant differences	M5-0-16, M5-0-17, A5-2-5	
CTR56-CPP. Do not use pointer arithmetic on polymorphic objects.	2 - Small differences	A5-0-4	
CTR57-CPP. Provide a valid ordering predicate.	2 - Small differences	A25-4-1	
CTR58-CPP. Predicate function objects should not be mutable.	2 - Small differences	A25-1-1	
ARR30-C. Do not form or use out-of-bounds pointers or array subscripts.	3 - Significant differences	A5-2-5	
ARR37-C. Do not add or subtract an integer to a pointer to a non-array object.	3 - Significant differences	M5-0-15	
ARR38-C. Guarantee that library functions do not form invalid pointers.	3 - Significant differences	RuleLinkM5-0-16	
ARR39-C. Do not add or subtract a scaled integer to a pointer.	3 - Significant differences	M5-0-15	
STR30-C. Do not attempt to modify string literals.	4 - Rejected		Use of C-style arrays, apart from static constexpr members, is prohibited. See: A18-1-1.

STR31-C. Guarantee that storage for strings has sufficient space for character data and the null terminator.	3 - Significant differences	A5-2-5	Effectively covered by prohibiting out-of-range array access.
STR32-C. Do not pass a non-null-terminated character sequence to a library function that expects a string	4 - Rejected		Use of functions from `<cstring>` is prohibited by M18-0-5.
STR34-C. Cast characters to unsigned char before converting to larger integer sizes.	3 - Significant differences	M5-0-4	
STR37-C. Arguments to character-handling functions must be representable as an unsigned char.	2 - Small differences	A21-8-1	
STR38-C. Do not confuse narrow and wide character strings and functions.	3 - Significant differences	A2-13-2, A2-13-3	Use of `wchar_t` is prohibited.
STR50-CPP. Guarantee that storage for strings has sufficient space for character data and the null terminator.	3 - Significant differences	A5-2-5	Effectively covered by prohibiting out-of-range array access.
STR51-CPP. Do not attempt to create a std::string from a null pointer.	3 - Significant differences	A5-3-2	This is a special case of a more general rule.
STR52-CPP. Use valid references, pointers, and iterators to reference elements of a basic_string.	2 - Small differences	A23-0-2	
STR53-CPP. Range check element access.	3 - Significant differences	A5-2-5	The specific case of A5-2-5.
MEM31-C. Free dynamically allocated memory when no longer needed.	3 - Significant differences	A15-1-4, A18-5-2	Intent of this rule is covered by effectively demanding the use of the RAII pattern.
MEM34-C. Only free memory allocated dynamically.	4 - Rejected		Use of malloc, calloc and realloc functions is prohibited, see: A18-5-1.
MEM35-C. Allocate sufficient memory for an object.	4 - Rejected		Use of malloc, calloc and realloc functions is prohibited, see: A18-5-1.

MEM36-C. Do not modify the alignment of objects by calling realloc().	4 - Rejected		Use of malloc, calloc and realloc functions is prohibited, see: A18-5-1.
MEM50-CPP. Do not access freed memory.	2 - Small differences	A3-8-1	
MEM51-CPP. Properly deallocate dynamically allocated resources.	3 - Significant differences	A18-5-3	Use of memory allocation and deallocation operators limited by A18-5-2, A18-5-4.
MEM52-CPP. Detect and handle memory allocation errors.	3 - Significant differences	A15-0-2, A15-2-2, A15-3-3, A15-5-3	
MEM53-CPP. Explicitly construct and destruct objects when manually managing object lifetime.	4 - Rejected	A18-5-2	Explicit use of operators new and delete is prohibited. Managing object lifetime also covered by A18-5-1, A18-5-3.
MEM54-CPP. Provide placement new with properly aligned pointers to sufficient storage capacity.	2 - Small differences	A18-5-10	
MEM55-CPP. Honor replacement dynamic storage management requirements.	2 - Small differences	A18-5-9	
MEM56-CPP. Do not store an already-owned pointer value in an unrelated smart pointer.	2 - Small differences	A20-8-1	
MEM57-CPP. Avoid using default operator new for over-aligned types.	4 - Rejected		Current approach is to use managed memory objects, which does not allow to work on a raw storage, but on a type storage, thus no need for this rule.
FIO50-CPP. Do not alternately input and output from a file stream without an intervening positioning call.	2 - Small differences	A27-0-3	

FIO51-CPP. Close files when they are no longer needed.	4 - Rejected		Usage of RAII solves the problem and is recommended, see: A15-1-4, A18-5-2.
FIO30-C. Exclude user input from format strings.	2 - Small differences	A27-0-1	
FIO32-C. Do not perform operations on devices that are only appropriate for files.	4 - Rejected		Non-generic rule affecting only particular file types on some operating systems.
FIO34-C. Distinguish between characters read from a file and EOF or WEOF.	4 - Rejected	M27-0-1	The C IO library is not used.
FIO37-C. Do not assume that fgets() or fgetws() returns a nonempty string when successful.	4 - Rejected	M27-0-1	The C IO library is not used.
FIO38-C. Do not copy a FILE object.	4 - Rejected	M27-0-1	The C IO library is not used.
FIO39-C. Do not alternately input and output from a stream without an intervening flush or positioning call.	2 - Small differences	A27-0-3	
FIO40-C. Reset strings on fgets() or fgetws() failure.	4 - Rejected	M27-0-1	The C IO library is not used.
FIO41-C. Do not call getc(), putc(), getwc(), or putwc() with a stream argument that has side effects.	4 - Rejected	M27-0-1	The C IO library is not used.
FIO42-C. Close files when they are no longer needed.	4 - Rejected	M27-0-1	The C IO library is not used.
FIO44-C. Only use values for fsetpos() that are returned from fgetpos().	4 - Rejected	M27-0-1	The C IO library is not used.
FIO45-C. Avoid TOCTOU race conditions while accessing files.	5 - Not yet analyzed		The “Concurrency and Parallelism” chapter is not yet covered, this will be addressed in future. See also CP.2
FIO46-C. Do not access a closed file.	4 - Rejected	M27-0-1	The C IO library is not used.
FIO47-C. Use valid format strings.	4 - Rejected	M27-0-1	The C IO library is not used.

ERR30-C. Set errno to zero before calling a library function known to set errno, and check errno only after the function returns a value indicating failure.	4 - Rejected		Use of the errno is prohibited, see: M19-3-1.
ERR32-C. Do not rely on indeterminate values of errno.	4 - Rejected		Use of the errno is prohibited, see: M19-3-1.
ERR33-C. Detect and handle standard library errors.	3 - Small differences	M0-3-2, A15-0-3	
ERR34-C. Detect errors when converting a string to a number.	2 - Small differences	A18-0-2	
ERR50-CPP. Do not abruptly terminate the program.	2 - Small differences	A15-5-2, A15-5-3	
ERR51-CPP. Handle all exceptions.	2 - Small differences	A15-3-3, A15-5-3	
ERR52-CPP. Do not use setjmp() or longjmp().	2 - Small differences	M17-0-5	
ERR53-CPP. Do not reference base classes or class data members in a constructor or destructor function-try-block handler.	3 - Significant differences	M15-3-3	Use of function-try-blocks is anyway not recommended. See: A15-3-5.
ERR54-CPP. Catch handlers should order their parameter types from most derived to least derived.	2 - Small differences	M15-3-6, M15-3-7	
ERR55-CPP. Honor exception specifications.	3 - Significant differences	A15-4-2	Use of dynamic exception specification is prohibited, see: A15-4-1. The noexcept specifier should be used instead.
ERR56-CPP. Guarantee exception safety.	2 - Small differences	A15-0-2	
ERR57-CPP. Do not leak resources when handling exceptions.	3 - Significant differences	A15-0-2, A15-1-2, A15-1-4	
ERR58-CPP. Handle all exceptions thrown before main() begins executing.	2 - Small differences	A15-2-1	
ERR59-CPP. Do not throw an exception across execution boundaries.	2 - Small differences	A15-1-5	
ERR60-CPP. Exception objects must be nothrow copy constructible.	3 - Significant differences	A15-5-3	
ERR61-CPP. Catch exceptions by lvalue reference.	2 - Small differences	A15-3-5	
ERR62-CPP. Detect errors when converting a string to a number.	2 - Small differences	A18-0-2	

OOP50-CPP. Do not invoke virtual functions from constructors or destructors.	2 - Small differences	M12-1-1	
OOP51-CPP. Do not slice derived objects.	2 - Small differences	A12-8-6, A15-3-5	
OOP52-CPP. Do not delete a polymorphic object without a virtual destructor.	2 - Small differences	A12-4-1, A12-4-2	
OOP53-CPP. Write constructor member initializers in the canonical order.	2 - Small differences	A8-5-1	
OOP54-CPP. Gracefully handle self-copy assignment.	2 - Small differences	A12-8-5	
OOP55-CPP. Do not use pointer-to-member operators to access nonexistent members.	2 - Small differences	A5-5-1	
OOP56-CPP. Honor replacement handler requirements.	3 - Significant differences	A18-5-5, A15-5-2, A15-5-3, A15-4-1	std::terminate and std::unexpected handler forbidden, std::new_handler comments added.
OOP57-CPP. Prefer special member functions and overloaded operators to C Standard Library functions.	2 - Small differences	A12-0-2	
OOP58-CPP. Copy operations must not mutate the source object.	2 - Small differences	A12-8-1	
CON50-CPP. Do not destroy a mutex while it is locked.	5 - Not yet analyzed		The “Concurrency and Parallelism” chapter is not yet covered, this will be addressed in future.
CON51-CPP. Ensure actively held locks are released on exceptional conditions.	5 - Not yet analyzed		The “Concurrency and Parallelism” chapter is not yet covered, this will be addressed in future.
CON52-CPP. Prevent data races when accessing bit-fields from multiple threads.	5 - Not yet analyzed		The “Concurrency and Parallelism” chapter is not yet covered, this will be addressed in future.

CON53-CPP. Avoid deadlock by locking in a predefined order.	5 - Not yet analyzed		The “Concurrency and Parallelism” chapter is not yet covered, this will be addressed in future.
CON54-CPP. Wrap functions that can spuriously wake up in a loop.	5 - Not yet analyzed		The “Concurrency and Parallelism” chapter is not yet covered, this will be addressed in future.
CON55-CPP. Preserve thread safety and liveness when using condition variables.	5 - Not yet analyzed		The “Concurrency and Parallelism” chapter is not yet covered, this will be addressed in future.
CON56-CPP. Do not speculatively lock a non-recursive mutex that is already owned by the calling thread.	5 - Not yet analyzed		The “Concurrency and Parallelism” chapter is not yet covered, this will be addressed in future.
CON33-C. Avoid race conditions when using library functions.	5 - Not yet analyzed		The “Concurrency and Parallelism” chapter is not yet covered, this will be addressed in future.
CON37-C. Do not call signal() in a multithreaded program	4 - Rejected		Use of signal handling facilities of <csignal> is prohibited, see: M18-7-1.
CON40-C. Do not refer to an atomic variable twice in an expression.	5 - Not yet analyzed		The “Concurrency and Parallelism” chapter is not yet covered, this will be addressed in future.

CON41-C. Wrap functions that can fail spuriously in a loop.	5 - Not yet analyzed		The "Concurrency and Parallelism" chapter is not yet covered, this will be addressed in future.
CON43-C. Do not allow data races in multithreaded code.	5 - Not yet analyzed		The "Concurrency and Parallelism" chapter is not yet covered, this will be addressed in future.
MSC33-C. Do not pass invalid data to the asctime() function.	4 - Rejected		Use of time handling functions of <ctime> is prohibited, see: M18-0-4.
MSC38-C. Do not treat a predefined identifier as an object if it might only be implemented as a macro.	4 - Rejected		Error indicator errno, setjmp() and variadic arguments shall not be used, see: M19-3-1, M17-0-5, A8-4-1.
MSC39-C. Do not call va_arg() on a va_list that has an indeterminate value.	4 - Rejected		Use of variadic arguments is prohibited, see: A8-4-1.
MSC40-C. Do not violate constraints.	2 - Small differences	A1-1-1	
MSC50-CPP. Do not use std::rand() for generating pseudorandom numbers.	2 - Small differences	A26-5-1	
MSC51-CPP. Ensure your random number generator is properly seeded.	2 - Small differences	A26-5-2	
MSC52-CPP. Value-returning functions must return a value from all exit paths.	2 - Small differences	A8-4-2	
MSC53-CPP. Do not return from a function declared [[noreturn]].	2 - Small differences	A7-6-1	
MSC54-CPP. A signal handler must be a plain old function.	4 - Rejected		Use of signal handling facilities of <csignal> is prohibited, see: M18-7-1.
FLP30-C. Do not use floating-point variables as loop counters.	2 - Small differences	A6-5-2	
FLP32-C. Prevent or detect domain and range errors in math functions.	2 - Small differences	A0-4-4	
FLP34-C. Ensure that floating-point conversions are within range of the new type.	3 - Significant differences	M5-0-5, M5-0-6, M5-0-7	

FLP36-C. Preserve precision when converting integral values to floating-point type.	3 - Significant differences	M5-0-5, M5-0-6, M5-0-7	
FLP37-C. Do not use object representations to compare floating-point values.	2 - Small differences	M3-9-3	
ENV30-C. Do not modify the object referenced by the return value of certain functions.	2 - Small differences	M18-0-3, M18-0-4, A18-0-3, A17-1-1	Listed functions are prohibited by separate rules.
ENV31-C. Do not rely on an environment pointer following an operation that may invalidate it.	4 - Rejected		In general, a project shall not rely on environment-specific implementations.
ENV32-C. All exit handlers must return normally.	3 - Significant differences	A15-5-2, A15-5-3	
ENV33-C. Do not call system().	2 - Small differences		Covered by M18-0-3
ENV34-C. Do not store pointers returned by certain functions.	3 - Significant differences	A18-0-3, M19-3-1	
SIG31-C. Do not access shared objects in signal handlers.	4 - Rejected		Use of signal handling facilities of <csignal> is prohibited, see: M18-7-1.
SIG34-C. Do not call signal() from within interruptible signal handlers.	4 - Rejected		Use of signal handling facilities of <csignal> is prohibited, see: M18-7-1.
SIG35-C. Do not return from a computational exception signal handler.	4 - Rejected		Use of signal handling facilities of <csignal> is prohibited, see: M18-7-1.
PRE30-C. Do not create a universal character name through concatenation.	4 - Rejected		Forbidden by A2-3-1.
PRE31-C. Avoid side effects in arguments to unsafe macros.	3 - Significant differences		Defining function-like macros is prohibited, see: A16-0-1.

PRE32-C. Do not use preprocessor directives in invocations of function-like macros.			Defining function-like macros is prohibited, see: A16-0-1.

Table B.4: SEI CERT C++

B.5 Traceability to C++ Core Guidelines

The following table demonstrates the traceability to C++ Core Guidelines [11]. This is not considered as a reproduction, but a mean to compare the two standards.

Note that the copyright of C++ Core Guidelines allows a derivative work anyway.

C++ Core Guidelines Rule:	Relation type:	Related rule:	Comment:
P.1: Express ideas directly in code.	4 - Rejected		The rule is vague.
P.2: Write in ISO Standard C++.	2 - Small differences	A0-4-3	
P.3: Express intent.	4 - Rejected		The rule is vague.
P.4: Ideally, a program should be statically type safe.	3 - Significant differences		The rule is covered by: A5-2-1, A5-2-2, A5-2-4, M5-2-12, A8-5-2, A9-5-1
P.5: Prefer compile-time checking to run-time checking.	3 - Significant differences	M0-3-1	
P.6: What cannot be checked at compile time should be checkable at run time.	3 - Significant differences	A0-1-2, M0-3-2	
P.7: Catch run-time errors early.	3 - Significant differences	A0-1-2, M0-3-2, A5-2-5, A15-0-4, A15-0-5	
P.8: Don't leak any resources.	3 - Significant differences	A18-5-1, A18-5-2, A15-1-4	
P.9: Don't waste time or space.	3 - Significant differences	M0-1-1, A0-1-1, M0-1-8, M0-1-9	
P.10: Prefer immutable data to mutable data.	2 - Small differences	A7-1-1	
P.11: Encapsulate messy constructs, rather than spreading through the code.	4 - Rejected		The rule is vague.
I.1: Make interfaces explicit.	4 - Rejected		The rule is vague.

I.2 Avoid global variables.	3 - Significant differences	A3-3-2	A3-3-2 covers only the initialization of global objects.
I.3: Avoid singletons.	3 - Significant differences	A3-3-2	A3-3-2 covers only the initialization of static objects.
I.4: Make interfaces precisely and strongly typed.	2 - Small differences	A8-4-14	-
I.5: State preconditions (if any).	4 - Rejected		The rule is vague.
I.6: Prefer Expects() for expressing preconditions.	4 - Rejected		Expects() is not part of Language Standard.
I.7: State postconditions.	4 - Rejected		The rule is vague.
I.8: Prefer Ensures() for expressing postconditions.	4 - Rejected		Ensures() is not part of Language Standard.
I.9: If an interface is a template, document its parameters using concepts.	3 - Significant differences	A14-1-1	
I.10: Use exceptions to signal a failure to perform a required task.	2 - Small differences	A15-0-1	
I.11: Never transfer ownership by a raw pointer (T*).	2 - Small differences	A8-4-12	
I.12: Declare a pointer that must not be null as not_null.	4 - Rejected		The not_null is not part of Language Standard.
I.13: Do not pass an array as a single pointer.	2 - Small differences	M5-2-12	
I.22: Avoid complex initialization of global objects.	2 - Small differences	A3-3-2	Intent of the rule achieved by restriction to constant-initialized objects.
I.23: Keep the number of function arguments low.	4 - Rejected		AUTOSAR C++ Coding Guidelines does not define code metrics, see: A1-4-1.
I.24: Avoid adjacent unrelated parameters of the same type.	4 - Rejected		Design rule, requiring case-by-case reasoning to apply.
I.25: Prefer abstract classes as interfaces to class hierarchies.	2 - Small differences	A10-4-1	

I.26: If you want a cross-compiler ABI, use a C-style subset.	2 - Small differences	M1-0-2	-
F.1: “Package” meaningful operations as carefully named functions.	5 - Not yet analyzed	-	-
F.2: A function should perform a single logical operation.	5 - Not yet analyzed	-	-
F.3: Keep functions short and simple.	4 - Rejected		AUTOSAR C++ Coding Guidelines does not define code metrics, see: A1-4-1.
F.4: If a function may have to be evaluated at compile time, declare it constexpr.	2 - Small differences	A7-1-2	
F.5: If a function is very small and time-critical, declare it inline.	4 - Rejected		AUTOSAR C++ Coding Guidelines does not define code metrics, see: A1-4-1.
F.6: If your function may not throw, declare it noexcept.	2 - Small differences	A15-4-4	
F.7: For general use, take T* or T& arguments rather than smart pointers.	2 - Small differences	A8-4-11	Added as a reference.
F.8: Prefer pure functions.	4 - Rejected	-	There is no need for a new rule.
F.9: Unused parameters should be unnamed.	2 - Small differences	A0-1-4, A0-1-5	
F.15: Prefer simple and conventional ways of passing information.	2 - Small differences	A8-4-3	
F.16: For “in” parameters, pass cheaply-copied types by value and others by reference to const.	2 - Small differences	A8-4-7	
F.17: For “in-out” parameters, pass by reference to non-const.	2 - Small differences	A8-4-9	
F.18: For “consume” parameters, pass by X&& and std::move the parameter.	2 - Small differences	A8-4-5	
F.19: For “forward” parameters, pass by TP&& and only std::forward the parameter.	2 - Small differences	A8-4-6	
F.20: For “out” output values, prefer return values to output parameters.	2 - Small differences	A8-4-8	
F.21: To return multiple “out” values, prefer returning a tuple or struct.	3 - Significant differences	A8-4-4	Prefer to return as a tuple.
F.22: Use T* or owner<T*> to designate a single object.	3 - Significant differences	M5-2-12	The owner<T*> is not part of Language Standard.

F.23: Use a not_null<T> to indicate that "null" is not a valid value.	4 - Rejected		The not_null<T> is not part of Language Standard.
F.24: Use a span<T> or a span_p<T> to designate a half-open sequence.	4 - Rejected		Neither the span<T> nor the span_p<T> are part of Language Standard.
F.25: Use a zstring or a not_null<zstring> to designate a C-style string.	4 - Rejected		Neither the zstring nor the not_null<zstring> are part of Language Standard.
F.26: Use a unique_ptr<T> to transfer ownership where a pointer is needed.	2 - Small differences	A20-8-2	
F.27: Use a shared_ptr<T> to share ownership.	2 - Small differences	A20-8-3	
F.60: Prefer T* over T& when no argument is a valid option.	3 - Significant differences	A8-4-10	
F.42: Return a T* to indicate a position (only).	3 - Significant differences	M5-0-15, M5-0-16, M5-0-17, M5-0-18	
F.43: Never (directly or indirectly) return a pointer or a reference to a local object.	2 - Small differences	M7-5-2	Added as a note to this rule.
F.44: Return a T& when copy is undesirable and returning no object isn't needed.	3 - Significant differences	A9-3-1	Do not return non-const reference to private fields.
F.45: Don't return a T&&.	2 - Small differences	M7-5-2	Added as a reference to this rule.
F.46: int is the return type for main().	2 - Small differences	A0-4-3	Added as a reference to this rule.
F.47: Return T& from assignment operators.	2 - Small differences	A13-2-1	Added as a reference to this rule.

F.50: Use a lambda when a function won't do (to capture local variables, or to write a local function).	4 - Rejected		Design principle. Prefer using lambda over functions while variables capture is required or it is defined in a local scope.
F.51: Where there is a choice, prefer default arguments over overloading.	4 - Rejected		Design principle. Prefer default arguments over overloading.
F.52: Prefer capturing by reference in lambdas that will be used locally, including passed to algorithms.	4 - Rejected		This rule is too specific and this is a matter of ownership, not only lambda.
F.53: Avoid capturing by reference in lambdas that will be used nonlocally, including returned, stored on the heap, or passed to another thread.	2 - Small differences	A5-1-4	
F.54: If you capture this, capture all variables explicitly (no default capture).	3 - Significant differences	A5-1-2	AUTOSAR C++ Coding Guidelines prohibits implicit variables capturing into a lambda expression.
F.55: Don't use va_arg arguments.	2 - Small differences	A8-4-1	Added as a reference to this rule.
C.1: Organize related data into structures (structs or classes).	4 - Rejected		Design principle. There is no need for a new rule..
C.2: Use class if the class has an invariant; use struct if the data members can vary independently.	3 - Significant differences		Class shall be used for all non-POD types (see: A11-0-1), and a struct for types defined in A11-0-2.
C.3: Represent the distinction between an interface and an implementation using a class.	4 - Rejected		This rule is vague.
C.4: Make a function a member only if it needs direct access to the representation of a class.	2 - Small differences	M9-3-3	

C.5: Place helper functions in the same namespace as the class they support.	5 - Not yet analyzed	-	-
C.7: Don't define a class or enum and declare a variable of its type in the same statement.	2 - Small differences	A7-1-9	
C.8: Use class rather than struct if any member is non-public.	2 - Small differences	M11-0-1, A11-0-1	
C.9: Minimize exposure of members.	3 - Significant differences	M9-3-1, A9-3-1, M11-0-1	
C.10 Prefer concrete types over class hierarchies.	4 - Rejected		This rule is vague. Concrete types and class hierarchies are highly context dependent.
C.11: Make concrete types regular.	4 - Rejected		This rule is vague. "Concrete types" has no technical meaning.
C.20: If you can avoid defining default operations, do.	2 - Small differences	A12-0-1	Following "the rule of zero" is permitted if no special member functions need to be defined.
C.21: If you define or =delete any default operation, define or =delete them all.	2 - Small differences	A12-0-1	
C.22: Make default operations consistent.	2 - Small differences	A12-1-1, A12-8-1	
C.30: Define a destructor if a class needs an explicit action at object destruction.	4 - Rejected		Design principle. Destructor shall be defined only if a class needs an explicit action at object destruction.
C.31: All resources acquired by a class must be released by the class's destructor.	4 - Rejected		Implementation principle. There is no need for a new rule.
C.32: If a class has a raw pointer (T*) or reference (T&), consider whether it might be owning.	4 - Rejected		Memory managing objects are recommended.
C.33: If a class has an owning pointer member, define a destructor.	4 - Rejected		Memory managing objects are recommended.

C.34: If a class has an owning reference member, define a destructor.	4 - Rejected		Memory managing objects are recommended.
C.35: A base class destructor should be either public and virtual, or protected and nonvirtual.	2 - Small differences	A12-4-1	
C.36: A destructor may not fail.	2 - Small differences	A15-5-1	
C.37: Make destructors noexcept.	2 - Small differences	A15-5-1	
C.40: Define a constructor if a class has an invariant.	4 - Rejected		Design principle. There is no need for a new rule.
C.41: A constructor should create a fully initialized object.	2 - Small differences	A15-2-2, A12-1-1, A8-5-0	
C.42: If a constructor cannot construct a valid object, throw an exception.	2 - Small differences	A15-2-2	
C.43: Ensure that a class has a default constructor.	4 - Rejected		This rule is vague.
C.44: Prefer default constructors to be simple and non-throwing.	4 - Rejected		Non-generic design principle; There is no need for a new rule.
C.45: Don't define a default constructor that only initializes data members; use in-class member initializers instead.	2 - Small differences	A12-1-3, A12-7-1	
C.46: By default, declare single-argument constructors explicit.	2 - Small differences	A12-1-4	
C.47: Define and initialize member variables in the order of member declaration.	2 - Small differences	A8-5-1	
C.48: Prefer in-class initializers to member initializers in constructors for constant initializers.	3 - Significant differences	A12-1-3	AUTOSAR C++ Coding Guidelines states that NSDMI shall not be mixed with member initializer list of constructors, see: A12-1-2.
C.49: Prefer initialization to assignment in constructors.	2 - Small differences	A12-6-1	
C.50: Use a factory function if you need "virtual behavior" during initialization.	3 - Significant differences	M12-1-1	Added as a reference to this rule, as it has a good example of virtual behaviour during initialization.

C.51: Use delegating constructors to represent common actions for all constructors of a class.	2 - Small differences	A12-1-5	
C.52: Use inheriting constructors to import constructors into a derived class that does not need further explicit initialization.	2 - Small differences	A12-1-6	
C.60: Make copy assignment non-virtual, take the parameter by const&, and return by non-const&.	2 - Small differences	A10-3-5, A13-2-1	
C.61: A copy operation should copy.	2 - Small differences	A12-8-1, A12-8-2	
C.62: Make copy assignment safe for self-assignment.	2 - Small differences	A12-8-5	
C.63: Make move assignment non-virtual, take the parameter by &&, and return by non-const&.	2 - Small differences	A10-3-5, A13-2-1	
C.64: A move operation should move and leave its source in a valid state.	2 - Small differences	A12-8-1, A12-8-4	
C.65: Make move assignment safe for self-assignment.	2 - Small differences	A12-8-5	
C.66: Make move operations noexcept.	2 - Small differences	A15-5-1	
C.67: A base class should suppress copying, and provide a virtual clone instead if copying" is desired.	2 - Small differences	A12-8-6	
C.80: Use =default if you have to be explicit about using the default semantics.	2 - Small differences	A12-7-1	
C.81: Use =delete when you want to disable default behavior (without wanting an alternative).	2 - Small differences	A12-0-1, A12-8-6	
C.82: Don't call virtual functions in constructors and destructors.	2 - Small differences	M12-1-1	
C.83: For value-like types, consider providing a noexcept swap function.	3 - Significant differences	A12-8-2	The swap function is explicitly recommended for copy and move assignment operators only.
C.84: A swap function may not fail.	2 - Small differences	A15-5-1	
C.85: Make swap noexcept.	2 - Small differences	A15-5-1	
C.86: Make == symmetric with respect to operand types and noexcept.	2 - Small differences	A13-5-5	-
C.87: Beware of == on base classes.	2 - Small differences	A13-5-5	This rule is implicitly covered by adhering to A13-5-5.
C.89: Make a hash noexcept.	2 - Small differences	A18-1-6	

C.120: Use class hierarchies to represent concepts with inherent hierarchical structure (only).	4 - Rejected		Non-generic design principle; There is no need for a new rule.
C.121: If a base class is used as an interface, make it a pure abstract class.	2 - Small differences		AUTOSAR C++ Coding Guidelines defines an interface class definition, see: Interface-Class.
C.122: Use abstract classes as interfaces when complete separation of interface and implementation is needed.	2 - Small differences	A10-4-1	
C.126: An abstract class typically doesn't need a constructor.	4 - Rejected		Non-generic design principle; There is no need for a new rule.
C.127: A class with a virtual function should have a virtual or protected destructor.	2 - Small differences	A12-4-1	
C.128: Virtual functions should specify exactly one of virtual, override, or final.	2 - Small differences	A10-3-1	
C.129: When designing a class hierarchy, distinguish between implementation inheritance and interface inheritance.	4 - Rejected		Non-generic design principle; There is no need for a new rule.
C.130: Redefine or prohibit copying for a base class; prefer a virtual clone function instead.	2 - Small differences	A12-8-6	
C.131: Avoid trivial getters and setters.	4 - Rejected		All members in non-POD types shall be private.
C.132: Don't make a function virtual without reason.	4 - Rejected		Non-generic design principle; There is no need for a new rule.
C.133: Avoid protected data.	3 - Significant differences	M11-0-1	All members in non-POD types shall be private.
C.134: Ensure all non-const data members have the same access level.	2 - Small differences	M11-0-1, A11-0-2	

C.135: Use multiple inheritance to represent multiple distinct interfaces.	2 - Small differences	A10-1-1	
C.136: Use multiple inheritance to represent the union of implementation attributes.	4 - Rejected		Multiple implementation inheritance is prohibited by AUTOSAR C++ Coding Guidelines, it allows only multiple interface inheritance.
C.137: Use virtual bases to avoid overly general base classes.	4 - Rejected		It is allowed to use virtual inheritance only in a diamond hierarchy, see: M10-1-1, M10-1-2.
C.138: Create an overload set for a derived class and its bases with using.	4 - Rejected		Design principle. There is no need for a new rule.
C.139: Use final sparingly.	3 - Significant differences	A12-4-2	Class shall be declared final if it has a non-virtual destructor.
C.140: Do not provide different default arguments for a virtual function and an overrider.	2 - Small differences	M8-3-1	
C.145: Access polymorphic objects through pointers and references.	3 - Significant differences	A12-8-6, A15-3-5	Functionalities that could lead to slicing are prohibited.
C.146: Use dynamic_cast where class hierarchy navigation is unavoidable.	2 - Small differences	A5-2-1	
C.147: Use dynamic_cast to a reference type when failure to find the required class is considered an error.	4 - Rejected		The dynamic_cast should not be used, see: A5-2-1.
C.148: Use dynamic_cast to a pointer type when failure to find the required class is considered a valid alternative.	4 - Rejected		The dynamic_cast should not be used, see: A5-2-1.
C.149: Use unique_ptr or shared_ptr to avoid forgetting to delete objects created using new.	2 - Small differences	A18-5-2	
C.150: Use make_unique() to construct objects owned by unique_ptrs.	2 - Small differences	A20-8-5	
C.151: Use make_shared() to construct objects owned by shared_ptrs.	2 - Small differences	A20-8-6	

C.152: Never assign a pointer to an array of derived class objects to a pointer to its base.	2 - Small differences	M5-2-12	
C.153: Prefer virtual function to casting.	3 - Significant differences	M5-2-2, M5-2-3, A5-2-1	
C.160: Define operators primarily to mimic conventional usage.	4 - Rejected		Design principle. There is no need for a new rule.
C.161: Use nonmember functions for symmetric operators.	4 - Rejected		Design principle. There is no need for a new rule.
C.162: Overload operations that are roughly equivalent.	4 - Rejected		Design principle. There is no need for a new rule.
C.163: Overload only for operations that are roughly equivalent.	4 - Rejected		Design principle. There is no need for a new rule.
C.164: Avoid conversion operators.	2 - Small differences	A13-5-3	
C.165: Use using for customization points.	4 - Rejected		Implementation principle. There is no need for a new rule.
C.166: Overload unary & only as part of a system of smart pointers and references.	3 - Significant differences	M5-3-3	The unary & operator shall not be overloaded.
C.167: Use an operator for an operation with its conventional meaning.	4 - Rejected		Design principle; There is no need for a new rule.
C.168: Define overloaded operators in the namespace of their operands.	4 - Rejected		Design principle. There is no need for a new rule.
C.170: If you feel like overloading a lambda, use a generic lambda.	4 - Rejected		Design principle; There is no need for a new rule. Creating generic lambda expressions is allowed, see: A7-1-5.

C.180: Use unions to save memory.	4 - Rejected		Unions shall not be used, see: A9-5-1.
C.181: Avoid “naked” unions.	2 - Small differences	A9-5-1	
C.182: Use anonymous unions to implement tagged unions.	2 - Small differences	A9-5-1	Tagged unions allowed as an exception, but only as POD.
C.183: Don’t use a union for type punning.	4 - Rejected		Unions shall not be used, see: A9-5-1.
Enum.1: Prefer enumerations over macros.	3 - Significant differences	A16-0-1	Usage of macros is prohibited.
Enum.2: Use enumerations to represent sets of related named constants.	2 - Small differences	A7-2-5	
Enum.3: Prefer class enums over “plain” enums.	2 - Small differences	A7-2-3	
Enum.4: Define operations on enumerations for safe and simple use.	4 - Rejected		Design principle. There is no need for a new rule.
Enum.5: Don’t use ALL_CAPS for enumerators.	4 - Rejected		AUTOSAR C++ Coding Guidelines does not introduce rules related to coding style or naming convention.
Enum.6: Avoid unnamed enumerations.	3 - Significant differences	A7-2-3	Enum classes shall be used instead of enums; it is not allowed to declare unnamed enum class.
Enum.7: Specify the underlying type of an enumeration only when necessary.	4 - Rejected		AUTOSAR C++ Coding Guidelines forces a programmer to specify the underlying base type explicitly, as only fixed-width numeric types shall be used. See: A3-9-1.

Enum.8: Specify enumerator values only when necessary.	3 - Significant differences	A7-2-4	It is defined how enumerators values should be specified.
R.1: Manage resources automatically using resource handles and RAII (Resource Acquisition Is Initialization).	4 - Rejected		AUTOSAR C++ Coding Guidelines does not define rules for coding patterns. Note that usage of RAII is recommended, see: A15-1-4, A18-5-2.
R.2: In interfaces, use raw pointers to denote individual objects (only).	2 - Small differences	M5-2-12	
R.3: A raw pointer (a T*) is non-owning.	4 - Rejected		Ownership is covered by memory managing objects, see: A18-5-2.
R.4: A raw reference (a T&) is non-owning.	4 - Rejected		Ownership is covered by memory managing objects, see: A18-5-2.
R.5: Don't heap-allocate unnecessarily.	2 - Small differences	A18-5-8	
R.6: Avoid non-const global variables.	3 - Significant differences	A3-3-2	A3-3-2 covers only the initialization of global objects.
R.10: Avoid malloc() and free().	2 - Small differences	A18-5-1	
R.11: Avoid calling new and delete explicitly.	2 - Small differences	A18-5-2	
R.12: Immediately give the result of an explicit resource allocation to a manager object.	2 - Small differences	A18-5-2	
R.13: Perform at most one explicit resource allocation in a single expression statement.	2 - Small differences	A5-0-1	
R.14: ??? array vs. pointer parameter.	2 - Small differences	M5-2-12	
R.15: Always overload matched allocation/deallocation pairs.	3 - Significant differences	A18-5-3	
R.20: Use unique_ptr or shared_ptr to represent ownership.	2 - Small differences	A20-8-2, A20-8-3	
R.21: Prefer unique_ptr over shared_ptr unless you need to share ownership.	2 - Small differences	A20-8-4	
R.22: Use make_shared() to make shared_ptrs.	2 - Small differences	A20-8-6	

R.23: Use make_unique() to make unique_ptrs.	2 - Small differences	A20-8-5	
R.24: Use std::weak_ptr to break cycles of shared_ptrs.	2 - Small differences	A20-8-7	
R.30: Take smart pointers as parameters only to explicitly express lifetime semantics.	2 - Small differences	A8-4-11	
R.31: If you have non-std smart pointers, follow the basic pattern from std.	4 - Rejected		There is no need for a new rule. Smart pointers are a part of Language Standard.
R.32: Take a unique_ptr<widget> parameter to express that a function assumes ownership of a widget.	2 - Small differences	A8-4-12	
R.33: Take a unique_ptr<widget>& parameter to express that a function reseats thewidget.	2 - Small differences	A8-4-12	
R.34: Take a shared_ptr<widget> parameter to express that a function is part owner.	2 - Small differences	A8-4-13	
R.35: Take a shared_ptr<widget>& parameter to express that a function might reseat the shared pointer.	2 - Small differences	A8-4-13	
R.36: Take a const shared_ptr<widget>& parameter to express that it might retain a reference count to the object ???.	2 - Small differences	A8-4-13	
R.37: Do not pass a pointer or reference obtained from an aliased smart pointer.	2 - Small differences	A8-4-11, A18-5-2	Added as a reference in these rules and a note in A8-4-11.
ES.1: Prefer the standard library to other libraries and to “handcrafted code”.	4 - Rejected		Design principle; There is no need for a new rule.
ES.2: Prefer suitable abstractions to direct use of language features.	4 - Rejected		Design principle; There is no need for a new rule.
ES.5: Keep scopes small.	2 - Small differences	M3-4-1	
ES.6: Declare names in for-statement initializers and conditions to limit scope.	2 - Small differences	M3-4-1	As an exception from the A7-1-7, it is allowed to declare variables in for-statement initializer.

ES.7: Keep common and local names short, and keep uncommon and nonlocal names longer.	4 - Rejected		AUTOSAR C++ Coding Guidelines does not introduce rules related to coding style or naming convention.
ES.8: Avoid similar-looking names.	2 - Small differences	M2-10-1	
ES.9: Avoid ALL_CAPS names.	4 - Rejected		AUTOSAR C++ Coding Guidelines does not introduce rules related to coding style or naming convention.
ES.10: Declare one name (only) per declaration.	2 - Small differences	A7-1-7	
ES.11: Use auto to avoid redundant repetition of type names.	3 - Significant differences	A7-1-5	It is not recommended to use the auto specifier, but it is allowed.
ES.12: Do not reuse names in nested scopes.	2 - Small differences	A2-10-1	
ES.20: Always initialize an object.	2 - Small differences	A8-5-0	
ES.21: Don't introduce a variable (or constant) before you need to use it.	2 - Small differences	M3-4-1	
ES.22: Don't declare a variable until you have a value to initialize it with.	2 - Small differences	M3-4-1, A8-5-0	
ES.23: Prefer the {} initializer syntax.	2 - Small differences	A8-5-2	
ES.24: Use a unique_ptr<T> to hold pointers.	3 - Significant differences	A18-5-2, A15-1-4, A18-1-3	AUTOSAR C++ Coding Guidelines does not force a programmer to use std::unique_ptr, it is just highly recommended within examples and rationales.
ES.25: Declare an object const or constexpr unless you want to modify its value later on.	2 - Small differences	A7-1-1	
ES.26: Don't use a variable for two unrelated purposes.	4 - Rejected		This rule is vague. "Unrelated" is highly context dependent.

ES.27: Use std::array or stack_array for arrays on the stack.	3 - Significant differences	A18-1-1	C-style arrays shall not be used, and it is recommended to use std::array instead.
ES.28: Use lambdas for complex initialization, especially of const variables.	4 - Rejected		Design principle. There is no need for a new rule.
ES.30: Don't use macros for program text manipulation.	2 - Small differences	A16-0-1	Usage of macros is prohibited.
ES.31: Don't use macros for constants or "functions".	2 - Small differences	A16-0-1	Usage of macros is prohibited.
ES.32: Use ALL_CAPS for all macro names.	4 - Rejected		AUTOSAR C++ Coding Guidelines does not introduce rules related to coding style or naming convention.
ES.33: If you must use macros, give them unique names.	2 - Small differences	M2-10-1	
ES.34: Don't define a (C-style) variadic function.	2 - Small differences	A8-4-1	
ES.70: Prefer a switch-statement to an if-statement when there is a choice.	4 - Rejected		Design principle; The switch statement shall have at least two case-clauses, distinct from the default label. See: A6-4-1.
ES.71: Prefer a range-for-statement to a for-statement when there is a choice.	3 - Significant differences	A6-5-1	It is recommended to use range-based for statement to replace equivalent for-statements.
ES.72: Prefer a for-statement to a while-statement when there is an obvious loop variable.	2 - Small differences	A6-5-2	
ES.73: Prefer a while-statement to a for-statement when there is no obvious loop variable.	3 - Significant differences	A6-5-2	It is required that a for-loop contains a loop-counter.

ES.74: Prefer to declare a loop variable in the initializer part of a for-statement.	3 - Significant differences	M3-4-1	It is required that each identifier is defined in a block that minimizes its visibility.
ES.75: Avoid do-statements.	2 - Small differences	A6-5-3	
ES.76: Avoid goto.	2 - Small differences	A6-6-1	
ES.78: Always end a non-empty case with a break.	2 - Small differences	M6-4-5	
ES.79: Use default to handle common cases (only).	4 - Rejected		There is no need for a new rule.
ES.84: Don't (try to) declare a local variable with no name.	2 - Small differences	A6-2-2	-
ES.85: Make empty statements visible.	2 - Small differences	M6-3-1, M6-4-1, M6-4-1	
ES.86: Avoid modifying loop control variables inside the body of raw for-loops.	2 - Small differences	M6-5-3	
ES.40: Avoid complicated expressions.	4 - Rejected		This rule is vague. Order of evaluation is covered by A5-0-1.
ES.41: If in doubt about operator precedence, parenthesize.	2 - Small differences	A5-2-6, M5-0-2	
ES.42: Keep use of pointers simple and straightforward.	3 - Significant differences	M5-0-15, M5-0-16, A5-2-5	'span' is not covered as it is not part of the language standard.
ES.43: Avoid expressions with undefined order of evaluation.	2 - Small differences	A5-0-1	
ES.44: Don't depend on order of evaluation of function arguments.	2 - Small differences	A5-0-1	
ES.45: Avoid "magic constants"; use symbolic constants.	2 - Small differences	A5-1-1	
ES.46: Avoid lossy (narrowing, truncating) arithmetic conversions.	2 - Small differences	A4-7-1, M5-0-6	
ES.47: Use nullptr rather than 0 or NULL.	2 - Small differences	A4-10-1	
ES.48: Avoid casts.	2 - Small differences	A5-2-1, A5-2-2, A5-2-3, A5-2-4	
ES.49: If you must use a cast, use a named cast.	2 - Small differences	A5-2-2	
ES.50: Don't cast away const.	2 - Small differences	A5-2-3	
ES.55: Avoid the need for range checking.	3 - Significant differences	A6-5-1	A6-5-1 only covers for-loops

ES.56: Write std::move() only when you need to explicitly move an object to another scope.	3 - Significant differences	A12-8-3, A18-9-2, A18-9-3	Vulnerabilities of std::move() are explained.
ES.60: Avoid new and delete outside resource management functions.	2 - Small differences	A18-5-2	
ES.61: Delete arrays using delete[] and non-arrays using delete.	2 - Small differences	A18-5-3	
ES.62: Don't compare pointers into different arrays.	2 - Small differences	M5-0-16	
ES.63: Don't slice.	3 - Significant differences	A12-8-6, A15-3-5	The functionalities that could lead to slicing were prohibited.
ES.64: Use the T{e} notation for construction.	2 - Small differences	A8-5-2	
ES.65: Don't dereference an invalid pointer.	2 - Small differences	A5-3-2	
ES.100: Don't mix signed and unsigned arithmetic.	2 - Small differences	M5-0-4, M5-0-9	
ES.101: Use unsigned types for bit manipulation.	2 - Small differences	M5-0-21	
ES.102: Use signed types for arithmetic.	3 - Significant differences	A4-7-1, M5-19-1	
ES.103: Don't overflow.	2 - Small differences	A4-7-1	
ES.104: Don't underflow.	2 - Small differences	A4-7-1	
ES.105: Don't divide by zero.	2 - Small differences	A5-6-1	
Per.1: Don't optimize without reason.	4 - Rejected		Implementation principle; There is no need for a new rule.
Per.2: Don't optimize prematurely.	4 - Rejected		Implementation principle; There is no need for a new rule.
Per.3: Don't optimize something that's not performance critical.	4 - Rejected		Implementation principle; There is no need for a new rule.
Per.4: Don't assume that complicated code is necessarily faster than simple code.	4 - Rejected		Implementation principle; There is no need for a new rule.
Per.5: Don't assume that low-level code is necessarily faster than high-level code.	4 - Rejected		Implementation principle; There is no need for a new rule.

Per.6: Don't make claims about performance without measurements.	4 - Rejected		Implementation principle; There is no need for a new rule.
Per.7: Design to enable optimization.	4 - Rejected		Design principle; There is no need for a new rule.
Per.10: Rely on the static type system.	4 - Rejected		Implementation principle; There is no need for a new rule.
Per.11: Move computation from run time to compile time	2 - Small differences	A3-3-2, A7-1-1, A7-1-2	
Per.12: Eliminate redundant aliases	4 - Rejected		This rule is incomplete.
Per.13: Eliminate redundant indirections	4 - Rejected		This rule is incomplete.
Per.14: Minimize the number of allocations and deallocations	4 - Rejected		This rule is incomplete.
Per.15: Do not allocate on a critical branch	4 - Rejected		This rule is incomplete.
Per.16: Use compact data structures	4 - Rejected		This rule is incomplete.
Per.18: Space is time	4 - Rejected		This rule is incomplete.
Per.19: Access memory predictably.	4 - Rejected		Implementation principle. There is no need for a new rule.
CP.1: Assume that your code will run as part of a multi-threaded program.	5 - Not yet analyzed		The "Concurrency and Parallelism" chapter is not yet covered, this will be addressed in future.
CP.2: Avoid data races.	5 - Not yet analyzed		The "Concurrency and Parallelism" chapter is not yet covered, this will be addressed in future.

CP.3: Minimize explicit sharing of writable data.	5 - Not yet analyzed		The “Concurrency and Parallelism” chapter is not yet covered, this will be addressed in future.
CP.4: Think in terms of tasks, rather than threads.	5 - Not yet analyzed		The “Concurrency and Parallelism” chapter is not yet covered, this will be addressed in future.
CP.8: Don’t try to use volatile for synchronization.	3 - Significant differences	A2-11-1	Volatile keyword forbidden.
CP.20: Use RAII, never plain lock()/unlock().	5 - Not yet analyzed		The “Concurrency and Parallelism” chapter is not yet covered, this will be addressed in future.
CP.21: Use std::lock() to acquire multiple mutexes.	5 - Not yet analyzed		The “Concurrency and Parallelism” chapter is not yet covered, this will be addressed in future.
CP.22: Never call unknown code while holding a lock (e.g., a callback).	5 - Not yet analyzed		The “Concurrency and Parallelism” chapter is not yet covered, this will be addressed in future.
CP.23: Think of a joining thread as a scoped container.	5 - Not yet analyzed		The “Concurrency and Parallelism” chapter is not yet covered, this will be addressed in future.
CP.24: Think of a detached thread as a global container.	5 - Not yet analyzed		The “Concurrency and Parallelism” chapter is not yet covered, this will be addressed in future.

CP.25: Prefer gsl::raii_thread over std::thread unless you plan to detach().	5 - Not yet analyzed		The “Concurrency and Parallelism” chapter is not yet covered, this will be addressed in future.
CP.26: Prefer gsl::detached_thread over std::thread if you plan to detach().	5 - Not yet analyzed		The “Concurrency and Parallelism” chapter is not yet covered, this will be addressed in future.
CP.27: Use plain std::thread for threads that detach based on a run-time condition (only).	5 - Not yet analyzed		The “Concurrency and Parallelism” chapter is not yet covered, this will be addressed in future.
CP.28: Remember to join scoped threads that are not detach()ed.	5 - Not yet analyzed		The “Concurrency and Parallelism” chapter is not yet covered, this will be addressed in future.
CP.30: Do not pass pointers to local variables to non-raii_threads.	5 - Not yet analyzed		The “Concurrency and Parallelism” chapter is not yet covered, this will be addressed in future.
CP.31: Pass small amounts of data between threads by value, rather than by reference or pointer.	5 - Not yet analyzed		The “Concurrency and Parallelism” chapter is not yet covered, this will be addressed in future.
[CP.32: To share ownership between unrelated threads use shared_ptr.	5 - Not yet analyzed		The “Concurrency and Parallelism” chapter is not yet covered, this will be addressed in future.

CP.40: Minimize context switching.	5 - Not yet analyzed		The “Concurrency and Parallelism” chapter is not yet covered, this will be addressed in future.
CP.41: Minimize thread creation and destruction.	55 - Not yet analyzed		The “Concurrency and Parallelism” chapter is not yet covered, this will be addressed in future.
CP.42: Don’t wait without a condition.	5 - Not yet analyzed		The “Concurrency and Parallelism” chapter is not yet covered, this will be addressed in future.
CP.43: Minimize time spent in a critical section.	5 - Not yet analyzed		The “Concurrency and Parallelism” chapter is not yet covered, this will be addressed in future.
CP.44: Remember to name your lock_guards and unique_locks.	2 - Small differences	A6-2-2	This is a special case of the more generic rule.
P.50: Define a mutex together with the data it guards.	5 - Not yet analyzed		The “Concurrency and Parallelism” chapter is not yet covered, this will be addressed in future.
CP.60: Use a future to return a value from a concurrent task.	5 - Not yet analyzed		The “Concurrency and Parallelism” chapter is not yet covered, this will be addressed in future.
CP.61: Use a async() to spawn a concurrent task.	5 - Not yet analyzed		The “Concurrency and Parallelism” chapter is not yet covered, this will be addressed in future.

CP.100: Don't use lock-free programming unless you absolutely have to.	5 - Not yet analyzed		The "Concurrency and Parallelism" chapter is not yet covered, this will be addressed in future.
CP.101: Distrust your hardware/compiler combination.	5 - Not yet analyzed		The "Concurrency and Parallelism" chapter is not yet covered, this will be addressed in future.
CP.102: Carefully study the literature.	5 - Not yet analyzed		The "Concurrency and Parallelism" chapter is not yet covered, this will be addressed in future.
CP.110: Do not write your own double-checked locking for initialization.	5 - Not yet analyzed		The "Concurrency and Parallelism" chapter is not yet covered, this will be addressed in future.
CP.111: Use a conventional pattern if you really need double-checked locking.	5 - Not yet analyzed		The "Concurrency and Parallelism" chapter is not yet covered, this will be addressed in future.
CP.200: Use volatile only to talk to non-C++ memory.	3 - Significant differences	A2-11-1	Volatile keyword forbidden.
E.1: Develop an error-handling strategy early in a design.	4 - Rejected		Design principle; There is no need for a new rule.
E.2: Throw an exception to signal that a function can't perform its assigned task.	2 - Small differences	A15-0-1	
E.3: Use exceptions for error handling only.	2 - Small differences	A15-0-1	
E.4: Design your error-handling strategy around invariants.	4 - Rejected		Design principle; There is no need for a new rule.

E.5: Let a constructor establish an invariant, and throw if it cannot.	2 - Small differences	A15-2-2	
E.6: Use RAII to prevent leaks.	4 - Rejected		Design principle; There is no need for a new rule.
E.7: State your preconditions.	4 - Rejected		Design principle; There is no need for a new rule.
E.8: State your postconditions.	4 - Rejected		Design principle; There is no need for a new rule.
E.12: Use noexcept when exiting a function because of a throw is impossible or unacceptable.	2 - Small differences	A15-4-4	
E.13: Never throw while being the direct owner of an object.	3 - Significant differences	A15-1-4	It is required to release all acquired resources and objects before a throw or a return statement.
E.14: Use purpose-designed user-defined types as exceptions (not built-in types).	3 - Significant differences	A15-1-1	It is required that user-defined exceptions inherit from std::exception class.
E.15: Catch exceptions from a hierarchy by reference.	2 - Small differences	A15-3-5	
E.16: Destructors, deallocation, and swap must never fail.	2 - Small differences	A15-5-1	
E.17: Don't try to catch every exception in every function.	2 - Small differences	A15-3-2	AUTOSAR C++ Coding Guidelines introduces checked and unchecked exceptions. Whether an exception should be caught depends if meaningful action can be performed in a given context.

E.18: Minimize the use of explicit try/catch.	4 - Rejected		Implementation principle; There is no need for a new rule.
E.19: Use a final_action object to express cleanup if no suitable resource handle is available.	4 - Rejected		The finally is not part of the C++ Language Standard.
E.25: If you can't throw exceptions, simulate RAII for resource management.	3 - Rejected		the RAII is a coding pattern; There is no need for a new rule. On the other hand, usage of RAII is recommended in the example of the A15-1-4.
E.26: If you can't throw exceptions, consider failing fast.	4 - Rejected		Implementation principle; There is no need for a new rule.
E.27: If you can't throw exceptions, use error codes systematically.	4 - Rejected		AUTOSAR C++ Coding Guidelines does not force any specific error handling mechanism. It requires that every error information will be tested, see: M0-3-2.
E.28: Avoid error handling based on global state (e.g. errno).	2 - Small differences	M19-3-1	
Con.1: By default, make objects immutable.	2 - Small differences	A7-1-1	
Con.2: By default, make member functions const.	2 - Small differences	M9-3-3	
Con.3: By default, pass pointers and references to consts.	2 - Small differences	M7-1-2	
Con.4: Use const to define objects with values that do not change after construction.	2 - Small differences	A7-1-1	
Con.5: Use constexpr for values that can be computed at compile time.	2 - Small differences	A7-1-2	
T.1: Use templates to raise the level of abstraction of code.	4 - Rejected		Design principle; There is no need for a new rule.

T.2: Use templates to express algorithms that apply to many argument types.	4 - Rejected		Design principle; There is no need for a new rule.
T.3: Use templates to express containers and ranges.	4 - Rejected		Design principle; There is no need for a new rule.
T.4: Use templates to express syntax tree manipulation.	4 - Rejected		Design principle; There is no need for a new rule.
T.5: Combine generic and OO techniques to amplify their strengths, not their costs.	4 - Rejected		Design principle; There is no need for a new rule.
T.10: Specify concepts for all template arguments.	4 - Rejected		Concepts are not part of the C++14 Language Standard.
T.11: Whenever possible use standard concepts.	4 - Rejected		Concepts are not part of the C++14 Language Standard.
T.12: Prefer concept names over auto for local variables.	4 - Rejected		Concepts are not part of the C++14 Language Standard.
T.13: Prefer the shorthand notation for simple, single-type argument concepts.	4 - Rejected		Concepts are not part of the C++14 Language Standard.
T.20: Avoid “concepts” without meaningful semantics.	4 - Rejected		Concepts are not part of the C++14 Language Standard.
T.21: Require a complete set of operations for a concept.	4 - Rejected		Concepts are not part of the C++14 Language Standard.

T.22: Specify axioms for concepts.	4 - Rejected		Concepts are not part of the C++14 Language Standard.
T.23: Differentiate a refined concept from its more general case by adding new use patterns..	4 - Rejected		Concepts are not part of the C++14 Language Standard.
T.24: Use tag classes or traits to differentiate concepts that differ only in semantics..	4 - Rejected		Concepts are not part of the C++14 Language Standard.
T.25: Avoid complementary constraints.	4 - Rejected		Concepts are not part of the C++14 Language Standard.
T.26: Prefer to define concepts in terms of use-patterns rather than simple syntax.	4 - Rejected		Concepts are not part of the C++14 Language Standard.
T.40: Use function objects to pass operations to algorithms.	4 - Rejected		Implementation principle; There is no need for a new rule.
T.41: Require only essential properties in a template's concepts.	4 - Rejected		Concepts are not part of the C++14 Language Standard.
T.42: Use template aliases to simplify notation and hide implementation details.	4 - Rejected		Implementation principle. There is no need for a new rule.
T.43: Prefer using over typedef for defining aliases.	2 - Small differences	A7-1-6	
T.44: Use function templates to deduce class template argument types (where feasible).	4 - Rejected		Implementation principle; There is no need for a new rule.
T.46: Require template arguments to be at least Regular or SemiRegular.	4 - Rejected		Implementation principle; There is no need for a new rule.

T.47: Avoid highly visible unconstrained templates with common names.	4 - Rejected	M17-0-2, M17-0-3	This rule is vague. "Highly visible" and "common names" have no technical meaning.
T.48: If your compiler does not support concepts, fake them with enable_if.	4 - Rejected		Implementation principle; There is no need for a new rule.
T.49: Where possible, avoid type-erasure.	4 - Rejected		This rule is incomplete. This rule is vague. "Where possible" has no technical meaning.
T.60: Minimize a template's context dependencies.	4 - Rejected		This rule is vague.
T.61: Do not over-parameterize members (SCARY).	2 - Small differences	A14-1-1, A14-7-1	
T.62: Place non-dependent class template members in a non-templated base class.	2 - Small differences	A14-5-2	-
T.65: Use tag dispatch to provide alternative implementations of a function.	4 - Rejected		Implementation principle; There is no need for a new rule.
T.68: Use rather than () within templates to avoid ambiguities.	2 - Small differences	A8-5-2	
T.69: Inside a template, don't make an unqualified nonmember function call unless you intend it to be a customization point.	4 - Rejected		Implementation principle. There is no need for a new rule.
T.80: Do not naively templatize a class hierarchy.	5 - Not yet analyzed	-	-
T.81: Do not mix hierarchies and arrays.	2 - Small differences	A5-0-4	
T.83: Do not declare a member function template virtual.	4 - Rejected		There is no need for a new rule. Handled by a compiler.
T.84: Use a non-template core implementation to provide an ABI-stable interface.	4 - Rejected		Design principle. There is no need for a new rule.

T.100: Use variadic templates when you need a function that takes a variable number of arguments of a variety of types.	2 - Small differences	A8-4-1	
T.101: How to pass arguments to a variadic template.	4 - Rejected		This rule is incomplete.
T.102: How to process arguments to a variadic template.	4 - Rejected		There is no need for a new rule.
T.103: Don't use variadic templates for homogeneous argument lists.	4 - Rejected		There is no need for a new rule.
T.120: Use template metaprogramming only when you really need to.	4 - Rejected		This rule is vague. "When you really need to" has no technical meaning.
T.121: Use template metaprogramming primarily to emulate concepts.	4 - Rejected		Implementation principle. There is no need for a new rule.
T.122: Use templates (usually template aliases) to compute types at compile time.	4 - Rejected		Implementation principle. There is no need for a new rule.
T.123: Use constexpr functions to compute values at compile time.	4 - Rejected		Implementation principle. There is no need for a new rule.
T.124: Prefer to use standard-library TMP facilities.	4 - Rejected		There is no need for a new rule.
T.125: If you need to go beyond the standard-library TMP facilities, use an existing library.	4 - Rejected		This rule is vague. "beyond the standard-library TMP facilities" has no technical meaning.
T.140: Name all operations with potential for reuse.	4 - Rejected		
T.141: Use an unnamed lambda if you need a simple function object in one place only.	3 - Significant differences	A5-1-9	Only forbids duplication of identical lambda expressions.
T.142: Use template variables to simplify notation.	4 - Rejected		This rule is incomplete.

T.143: Don't write unintentionally nongeneric code.	5 - Not yet analyzed	-	-
T.144: Don't specialize function templates.	2 - Small differences	A14-8-2	
T.150: Check that a class matches a concept using static_assert.	2 - Small differences	A14-1-1	
CPL.1: Prefer C++ to C.	2 - Small differences	A17-1-1, A18-0-1	
CPL.2: If you must use C, use the common subset of C and C++, and compile the C code as C++.	4 - Rejected		There is no need for a new rule.
CPL.3: If you must use C for interfaces, use C++ in the calling code using such interfaces.	5 - Not yet analyzed	-	-
SF.1: Use a .cpp suffix for code files and .h for interface files if your project doesn't already follow another convention.	3 - Significant differences	A3-1-2, A3-1-3	For header file names, AUTOSAR C++ Coding Guidelines allows either ".h", ".hpp" or ".hxx" extension.
SF.2: A .h file may not contain object definitions or non-inline function definitions.	2 - Small differences	A3-1-1	
SF.3: Use .h files for all declarations used in multiple source files.	2 - Small differences	M3-2-2, A3-3-1	
SF.4: Include .h files before other declarations in a file.	2 - Small differences	M16-0-1	
SF.5: A .cpp file must include the .h file(s) that defines its interface.	5 - Not yet analyzed	-	-
SF.7: Don't write using namespace in a header file.	2 - Small differences	M7-3-6	
SF.8: Use #include guards for all .h files.	2 - Small differences	M16-2-3	
SF.9: Avoid cyclic dependencies among source files.	5 - Not yet analyzed	-	-
SF.21: Don't use an unnamed (anonymous) namespace in a header.	2 - Small differences	M7-3-3	
SF.22: Use an unnamed (anonymous) namespace for all internal/nonexported entities.	5 - Not yet analyzed	-	-
SL.1: Use libraries wherever possible.	4 - Rejected		Design principle; There is no need for a new rule.
SL.2: Prefer the standard library to other libraries.	4 - Rejected		Design principle; There is no need for a new rule.
SL.3: Do not add non-standard entities to namespace std.	2 - Small differences	A17-6-1	

SL.con.1: Prefer using STL array or vector instead of a C array.	2 - Small differences	A18-1-1	
SL.con.2: Prefer using STL vector by default unless you have a reason to use a different container.	4 - Rejected		This rule is vague. “have a reason to” has no technical meaning.
SL.str.1: Use std::string to own character sequences	2 - Small differences	A27-0-4	
SL.io.50: Avoid endl.	5 - Not yet analyzed	-	-
Type.1: Don’t use reinterpret_cast.	2 - Small differences	A5-2-4	
Type.2: Don’t use static_cast downcasts. Use dynamic_cast instead.	2 - Small differences	M5-2-2	
Type.3: Don’t use const_cast to cast away const (i.e., at all).	2 - Small differences	A5-2-3	
Type.4: Don’t use C-style (T) expression casts that would perform a static_cast downcast, const_cast, or reinterpret_cast.	2 - Small differences	A5-2-2	
Type.5: Don’t use a variable before it has been initialized.	2 - Small differences	A8-5-0	
Type.6: Always initialize a member variable.	2 - Small differences	A12-1-1	
Type.7: Avoid accessing members of raw unions. Prefer variant instead.	2 - Small differences	A9-5-1	
Type.8: Avoid reading from varargs or passing vararg arguments. Prefer variadic template parameters instead.	2 - Small differences	A8-4-1	
Bounds.1: Don’t use pointer arithmetic. Use span instead.	2 - Small differences	M5-0-15	
Bounds.2: Only index into arrays using constant expressions.	2 - Small differences	A5-2-5	
Bounds.3: No array-to-pointer decay.	2 - Small differences	M5-2-12	
Bounds.4: Don’t use standard library functions and types that are not bounds-checked.	5 - Not yet analyzed	-	-
NL.1: Don’t say in comments what can be clearly stated in code.	4 - Rejected		AUTOSAR C++ Coding Guidelines does not introduce rules related to coding style or naming convention.
NL.2: State intent in comments.	4 - Rejected		AUTOSAR C++ Coding Guidelines does not introduce rules related to coding style or naming convention.

NL.3: Keep comments crisp.	4 - Rejected		AUTOSAR C++ Coding Guidelines does not introduce rules related to coding style or naming convention.
NL.4: Maintain a consistent indentation style.	4 - Rejected		AUTOSAR C++ Coding Guidelines does not introduce rules related to coding style or naming convention.
NL.5 Don't encode type information in names.	4 - Rejected		AUTOSAR C++ Coding Guidelines does not introduce rules related to coding style or naming convention.
NL.7: Make the length of a name roughly proportional to the length of its scope.	4 - Rejected		AUTOSAR C++ Coding Guidelines does not introduce rules related to coding style or naming convention.
NL.8: Use a consistent naming style.	4 - Rejected		AUTOSAR C++ Coding Guidelines does not introduce rules related to coding style or naming convention.
NL.9: Use ALL_CAPS for macro names only.	4 - Rejected		AUTOSAR C++ Coding Guidelines does not introduce rules related to coding style or naming convention.

NL.10: Avoid CamelCase.	4 - Rejected		AUTOSAR C++ Coding Guidelines does not introduce rules related to coding style or naming convention.
NL.15: Use spaces sparingly.	4 - Rejected		AUTOSAR C++ Coding Guidelines does not introduce rules related to coding style or naming convention.
NL.16: Use a conventional class member declaration order.	4 - Rejected		AUTOSAR C++ Coding Guidelines does not introduce rules related to coding style or naming convention.
NL.17: Use K&R-derived layout.	4 - Rejected		AUTOSAR C++ Coding Guidelines does not introduce rules related to coding style or naming convention.
NL.18: Use C++-style declarator layout.	4 - Rejected		AUTOSAR C++ Coding Guidelines does not introduce rules related to coding style or naming convention.
NL.19: Avoid names that are easily misread.	2 - Small differences	M2-10-1	
NL.20: Don't place two statements on the same line.	3 - Significant differences	A7-1-7	It is required for declarations only.
NL.21: Declare one name (only) per declaration.	2 - Small differences	A7-1-7	
NL.25: Don't use void as an argument type.	5 - Not yet analyzed	-	-
NL.26: Use conventional const notation.	5 - Not yet analyzed	-	-

Table B.5: C++ Core Guidelines

B.6 Traceability to ISO 26262

Each method in the tables referenced from the ISO 26262 standard is an entry that is one of the following:

- consecutive – all methods shall be applied as recommended, but with different recommendation level that depends on the ASIL level.
- alternative – an appropriate combination of methods shall be applied.

Consecutive methods are marked by a sequence number, e.g. 1, 2, 3. Alternative entries are marked by a number followed by a letter, e.g. 1a, 1b, 1c.

This chapter traces mostly principles and recommendations from Section 8 (Software unit design and implementation) of the Part 6 of the ISO 26262. The rest of the document is not applicable to the AUTOSAR C++14 Coding Guidelines. Although AUTOSAR C++14 Coding Guidelines contain multiple rules allocated to other work products (e.g. software architectural design, verification), those are considered as recommendations for doing review of such work products, not as a direct fulfillment of ISO 26262 requirements.

Part 6: Product development at the software level

5: Initiation of product development at the software level

5.4: Requirements and recommendations

ISO 26262 requirement:		Relation type:	Related rule:	Comment:
1	an unambiguous definition	6 - Implemented	[3, ISO/IEC 14882:2014], A2-13-2, A1-1-1	Syntax and semantics of the language unambiguously defined. Restrictions on using conditionally-supported or deprecated features.
2	the support for embedded real time software and runtime error handling	6 - Implemented	A15-0-1, A15-0-4, A15-0-5, A15-0-6, A15-0-7, A15-0-8, A18-5-5, A18-5-6, A18-5-7	Introduction of Checked and Unchecked Exceptions. Usage of exceptions limited only to error handling. Additional requirements on toolchain support for memory management and exceptions handling.
3	the support for modularity, abstraction and structured constructs	6 - Implemented	A8-4-14, A10-0-1, A10-0-2, A10-4-1	Recommendations on which constructs to use to increase software reusability and hide implementation details.

Table B.6: The criteria that shall be considered when selecting a suitable modeling or programming language. Paragraph 5.4.6 from [5].

ISO 26262 requirement:		Relation type:	Related rule:	Comment:
1a	Enforcement of low complexity	6 - Implemented	A1-4-1	Requirements on code metrics. Plenty of AUTOSAR C++14 Coding Guidelines rules forbid constructs that introduce unnecessary complexity and are error-prone, e.g. A9-6-2, M10-2-1, A10-2-1.
1b	Use of language subsets	6 - Implemented	3.1	Plenty of AUTOSAR C++14 Coding Guidelines rules forbid constructs that are allowed from the C++ language perspective, but (1) lead to unstructured designs, (2) are misleading for a developer, (3) are implementation defined. In case some features are to be used in a particular project nonetheless, see chapter 5.4.
1c	Enforcement of strong typing	6 - Implemented	M5-2-2, M5-2-3, A5-2-2, A5-2-3, M5-2-6, A5-2-4, M5-2-9, A8-4-14, A7-2-3	Restrictions on type casting. Recommendations on strongly-typed interfaces and scoped enums.
1d	Use of defensive implementation techniques	6 - Implemented	A0-4-4, A4-7-1, A5-2-5, A6-5-1, A14-1-1, A15-3-4	Error checking required for math functions, integer expressions, array access. Limitations on iteration statements. Recommendations on how to cope with external code failures.
1e	Use of established design principles	6 - Implemented	6.18.5, A18-5-2, A0-1-4	Recommendation on RAII, exception in rules that facilitate correct usage of SFINAE and Concepts. Multiple rules contain references to corresponding rules from multiple standards, which confirms that the provided rule set reflects widely approved coding techniques.

1f	Use of unambiguous graphical representation	8 - Not applicable		Recommendations on graphical representation is out of scope of AUTOSAR C++14 Coding Guidelines.
1g	Use of style guides	8 - Not applicable		AUTOSAR C++14 Coding Guidelines does not introduce rules related to coding style.
1h	Use of naming conventions	8 - Not applicable		AUTOSAR C++14 Coding Guidelines does not introduce rules related to naming convention.

Table B.7: Topics to be covered by modelling and coding guidelines. Table 1 from [5].

8: Software unit design and implementation

8.4: Requirements and recommendations

ISO 26262 requirement:		Relation type:	Related rule:	Comment:
1a	Natural language	7 - Partially implemented	A2-7-3, A2-7-5	Requirements on: providing documentation for user-defined types, content and structure of the documentation.
1b	Informal notations			
1c	Semi-formal notations			
1d	Formal notations			

Table B.8: Notations for software unit design. Table 7 from [5].

ISO 26262 requirement:		Relation type:	Related rule:	Comment:
1a	One entry and one exit point in subprograms and functions	6 - Implemented		See `Single-point-of-exit`. All the benefits from this approach are handled by different measures. Code modularity aspects are also ensured by techniques and limitations that fulfill other ISO26262 requirements, see: Enforcement of low complexity, Readability and comprehensibility, Suitability for modifications.
1b	No dynamic objects or variables, or else online test during their creation	6 - Implemented	A18-5-5, A18-5-6, A18-5-7, A18-5-2, A18-5-9, A18-5-10	Dynamic memory allowed, but under multiple constraints affecting memory management. Allocated objects lifetime maintenance delegated to shared pointers and memory management objects. Restrictions set for custom implementations of dynamic memory allocation and placement new.

1c	Initialization of variables	6 - Implemented	A8-5-0, A3-3-2, A6-5-4, A8-5-1, A8-5-3, A8-5-4, A12-1-1, A12-1-2, A12-1-6	Required is: memory initialization before access, constant initialization of static and thread–local objects, loop counters initialization. Defined are requirements for: initialization lists, initialization of type auto, user-declared constructors, NSDMI, delegating constructors.
1d	No multiple use of variable names	6 - Implemented	M2-10-1, A2-10-1, A2-10-6, A2-10-4, A2-10-5, M3-4-1	Limitations on identifier hiding and requirements on typographical unambiguity.
1e	Avoid global variables or else justify their usage	7 - Partially implemented	A3-3-2	Global variables shall be constant-initialized.
1f	Limited use of pointers	6 - Implemented	A8-4-11, A8-4-10, M5-0-15, M5-0-16, M5-0-17, M5-0-18, A5-0-4, A5-0-3	Lifetime semantics is to be handled using smart pointers and usage of raw pointers is limited only to passing parameters that are not owned by the pointer (as currently no constructs from C++14 Standard could replace it), which resolves problems with possible memory leaks. Restrictions on pointers arithmetic, no more than two levels of pointer indirection, pointers usage limited only to single objects. Listed rule set fulfills also all the requirements described in [19], chapter C.2.6.6 “Limited use of pointers”, apart from “Data exchange should be done via the operating system”, which is out of scope of this document.
1g	No implicit type conversions	6 - Implemented	A4-5-1, A4-7-1, M4-10-1, M5-0-4, M5-0-5, M5-0-6, A7-2-3, A8-5-2, A13-5-2, A13-5-3, A23-0-1	It is forbidden to use: enums in arithmetic contexts, integer conversions that lead to data loss, NULL as an integer value, non-scoped enumerations, conversion from `iterator` to `const_iterator`. It is additionally protected by forcing: braced initialization of variables and restrictions on user-defined conversion operators.

1h	No hidden data or control flow	7 - Partially implemented	A15-0-6, A15-0-7, A15-0-8, 6.15	Usage of exceptions as a method for error handling is not enforced by the AUTOSAR Coding Guidelines. However, if it is to be used in a particular project, it (1) is allowed only if strict requirements on a toolchain are fulfilled, e.g. analysis of failure modes of exception handling, deterministic worst-case execution time. (2) is forbidden as part of the typical program flow and is limited only for handling errors where a function failed to perform its assigned task. (3) requires analysis of maximum execution time constraints for a particular software project. Using Checked Exceptions for signaling recoverable errors brings benefits to completeness of error handling procedure, as it enforces developers to provide a dedicated handling hook for each type of such errors. Rules for stack unwinding are unambiguously defined by [3, ISO/IEC 14882:2014], thus it is possible to establish a matching `catch` clause for each exception thrown in the software, to which the control flow will jump after exception is thrown. By default, handling of Unchecked Exceptions should lead to proper program termination, therefore stack unwinding ensures correct invocation of destructors for all objects with automatic storage duration, which prevents from possible memory leaks in case of errors (considering no violations of available rules for memory management). However, usage of exceptions still introduces a hidden control flow into a program execution, thus such an approach only partially implements this requirement from the ISO26262-6 [5].
1i	No unconditional jumps	6 - Implemented	A6-6-1, M17-0-2	Lack of exact definition of “unconditional jump” in ISO 26262 [5] and C++14 Standard [3], therefore the statement from IEC 61508 Annex C [19] was assumed: “avoid unconditional jumps (goto) in higher level languages”.
1j	No recursions	6 - Implemented	A7-5-2	

Table B.9: Design principles for software unit design and implementation. Table 8 from [5].

ISO 26262 requirement:		Relation type:	Related rule:	Comment:
1a	correct order of execution of subprograms and functions within the software units, based on the software architectural design	6 - Implemented	A15-4-1, A15-4-2, A15-5-2, A15-5-3	Restrictions on constructs that may lead to program termination without calling proper exit handlers and destructors.
1b	consistency of the interfaces between the software units	6 - Implemented	M1-0-2, A15-1-5	Restrictions on: passing non-standard layout type objects and exceptions across execution boundaries.
1c	correctness of data flow and control flow between and within the software units	6 - Implemented	M0-3-1 A15-0-1, A15-0-4, A15-0-5, A15-3-2, A15-3-3, A15-3-4, M15-0-3, A6-6-1	Checked and Unchecked exceptions are to be used only for error handling, they are forbidden for handling the normal control flow. Additional explanation of this property from [19]: “The software design shall include, commensurate with the required safety integrity level, self-monitoring of control flow and data flow. On failure detection, appropriate actions shall be taken.”
1d	simplicity	6 - Implemented	A1-4-1, A8-4-3, A10-1-1, A14-7-2	Forcing limitations for cases that limit maintainability and readability.
1e	readability and comprehensibility	6 - Implemented	A1-4-1, A1-4-3, A5-1-1, A5-1-9, A10-2-1, M10-2-1, A12-7-1, A14-7-2	Enforcing code metrics, forbidding constructs that are confusing, lead to code duplication or introduce unnecessary complexity.
1f	robustness	6 - Implemented	M0-3-1, M0-3-2, A1-1-1, A15-0-2, A15-0-3, A15-0-6, A15-0-7, A18-5-6	Checked exceptions concept which facilitates tracking if a particular error type is handled. Recommendations on how to report errors in an application. Requirements on analysis of failure modes and worst-case execution time for exception handling and memory management.

1g	suitability for software modification	6 - Implemented	A8-4-14, A10-0-1, A10-0-2, A10-4-1	Recommending constructs that increase software reusability, eliminate redundancy and hide implementation details.
1h	testability	6 - Implemented	A1-2-1, A1-4-1, A3-3-2	Requirements on: testing error information if this is generated, using code metrics which will increase code quality and modularity. Restrictions on constructs that obscure code maintainability. Suggestions on how to perform extensive testing.

Table B.10: Properties to be achieved by applying design principles from Table 8. Paragraph 8.4.4 from [5].

ISO 26262 requirement:		Relation type:	Related rule:	Comment:
1a	Walk-through	8 - Not applicable		Process of system examination to reveal discrepancies between a specification and implementation is out of scope of AUTOSAR C++14 Coding Guidelines.
1b	Inspection	8 - Not applicable		Structured processes (at any level of rigour) of revealing defects in developed software components are out of scope of AUTOSAR C++14 Coding Guidelines.
1c	Semi-formal verification	8 - Not applicable		Aspects of proving the correctness of a program basing on an abstract model are out of scope of AUTOSAR C++14 Coding Guidelines.
1d	Formal verification			
1e	Control flow analysis	7 - Partially implemented	M0-1-1, M0-1-2, M0-1-8, M0-1-9, M0-1-10, A0-1-3	Available are rules oriented on finding suspect areas of code (e.g. inaccessible code, infeasible paths), but analysis of directed program graph (from the definition from paragraph C.5.9 [19]) is out of scope of AUTOSAR C++14 Coding Guidelines.

1f	Data flow analysis	6 - Implemented	M0-1-3, M0-1-4, A0-1-1, A0-1-4, A0-1-5, A0-1-6, A8-5-0	Available are rules oriented on analysis of sequences of creating, referencing and deleting variables.
1g	Static code analysis	6 - Implemented		All automated rules included in AUTOSAR C++14 Coding Guidelines are enforceable by means of static code analysis.
1h	Semantic code analysis	8 - Not applicable		Mathematical source code analysis by use of an abstract representation of possible values is out of scope of AUTOSAR C++14 Coding Guidelines.

Table B.11: Methods for the verification of software unit design and implementation. Table 9 from [5].

C Glossary

Abbreviation / Acronym:	Description:
Real-time application (RTA)	A real-time application is a program that guarantees response within defined time constraints. The latency must be less than a defined value, usually measured in seconds or milliseconds. Whether or not a given application program qualifies as an RTA depends on the worst-case execution time (WCET) - the maximum length of time a defined task requires on a given hardware platform.
MISRA	Motor Industry Software Reliability Association.
HIC++	High Integrity C++ Coding Standard.
cvalue expression	An expression that should not undergo further conversions, either implicitly or explicitly, is called a cvalue expression.
Ownership	Ownership of a resource means that the resource's lifetime is fully managed by the single class instance or tied with the class instance lifetime. See also: chapter 6.18.5

One definition rule	The rule states that: • There shall be one and only one definition of any variable, function, class type, enumeration type, or template in a translation unit. Some of these may have multiple declarations, but only one definition is allowed. • There shall be one and only one definition of every non-inline function or variable that is odr-used in the entire program. • An inline function definition is required in every translation unit where it is odr-used. • There shall be one and only one definition of a class in any translation unit where the class is used in a way that requires it to be complete. • There can be more than one definition of any class, enumeration type, inline function with external linkage, class template, non-static function template, static data member of a class template, member function of a class template, partial template specialization in a program, as long as all of the following is true: – each definition consists of the same sequence of tokens (typically, appears in the same header file) – name lookup from within each definition finds the same entities (after overload-resolution), except that constants with internal or no linkage may refer to different objects as long as they are not ODR-used and have the same values in every definition. – overloaded operators, including conversion, allocation, and deallocation functions refer to the same function from each definition (unless referring to one defined within the definition) – the language linkage is the same (e.g. the include file isn't inside an extern "C" block) – the three rules above apply to every default argument used in each definition – if the definition is for a class with an implicitly-declared constructor, every translation unit where it is odr-used must call the same constructor for the base and members – if the definition is for a template, then all these requirements apply to both names at the point of definition and dependent names at the point of instantiation If all these requirements are satisfied, the program behaves as if there is only one definition in the entire program. Otherwise, the behavior is undefined.
ODR-use	An object is odr-used if its address is taken, or a reference is bound to it. A function is odr-used if a function call to it is made or its address is taken.

POD Type	POD (Plain Old Data) type is the type that is compatible with types used in the C programming language, can be manipulated using C library functions, and can be exchanged with C libraries directly in its binary form.
Trivially Copyable Class	A class (C++ Language Standard [3], chapter 9): • where each copy constructor, move constructor, copy assignment operator, move assigment operator is either deleted or trivial • that has at least one non-deleted copy constructor, move constructor, copy assignment operator, or move assignment operator, and • that has a trivial, non-deleted destructor
Standard-Layout Class	A class that (C++ Language Standard [3], chapter 9): • has no non-static data members of type non-standard-layout class (or array of such types) or reference • has no virtual functions and no virtual base classes • has the same access control for all non-static data members • has no non-standard-layout base classes • has at most one base class subobject of any given type • has all non-static data members and bit-fields in the class and its base classes first declared in the same class • has no element of the set M(X) of types as a base class where M(X) is defined as follows: – If X is a non-union class type, the set M(X) is empty if X has no (possibly inherited) non-static data members; otherwise, it consists of the type of the first non-static data member of X (where said member may be an anonymous union), X0, and the elements of M(X0). – If X is a union type, the set M(X) is the union of all M(Ui) and the set containing all Ui, where each Ui is the type of the i-th non-static data member of X. – If X is a non-class type, the set M(X) is empty.

Dataflow Anomaly	The state of a variable at a point in a program can be described using the following terms: • Undefined (U): The value of the variable is indeterminate. • Referenced (R): The variable is used in some way (e.g. in an expression). • Defined (D): The variable is explicitly initialized or assigned a value. Given the above, the following dataflow anomalies can be defined: • UR dataflow anomaly: Variable not assigned a value before the specified use (this may result in undefined behavior). • DU dataflow anomaly: Variable is assigned a value that is never subsequently used. • DD dataflow anomaly: Variable is assigned a value twice with no intermediate use.
Dead Code	Dead code (also known as redundant code) consists of evaluated expressions whose removal would not affect the output program.
Unreachable Code	Unreachable code is code to which there is no syntactic (control flow) path, e.g. a function which is never called, either directly or indirectly.
Diamond Problem	The “diamond problem” is an ambiguity that arises when two classes B and C inherit from A, and class D inherits from both B and C. If there is a method provided by class A, that is overriden in both B and C and D does not override it, then there is an ambiguity which version of the method does D actually inherit. See: Wikipedia.org for more details.
Interface class	An interface class is a class which has following properties: • if there are any, all member functions are public pure virtual • if there are any, all data members are public static constexpr
Extended precision format	The IEEE Standard for Floating-Point Arithmetic (IEEE 754) specifies extended precision formats, that are recommended for allowing a greater precision format than that provided by the basic formats. For an extended format the exponent range must be as great as that of the next wider basic format. For instance, 64-bit extended precision binary number must have an “exponent max” of at least 16383, which is equal to “exponent max” of 128-bit binary floating-point. The 80-bit extended format meets this requirement.

Fundamental types	C++ built-in types defined in C++ Language Standard [3] in chapter 3.9.1, e.g. char, signed char, unsigned char, int, long long int, wchar_t, bool, float, double, void, std::nullptr_t, etc.
Scalar types	The following types are scalar types: • integral types • floating point types • pointers and pointers to members • enumerations • std::nullptr_t
glvalue	A glvalue is an expression whose evaluation determines the identity of an object, bit-field, or function.
xvalue	An xvalue refers to an object, usually near the end of its lifetime, so that its resources may be moved.
prvalue	A prvalue is an expression whose evaluation initializes an object or a bit-field, or computes the value of the operand of an operator.
rvalue	An rvalue is an xvalue or a prvalue.
lvalue	An lvalue is a glvalue that is not an xvalue.
Implicitly-defined default constructor	Implicitly-defined default constructor calls default constructors of its base classes and non-static data members. It has exactly the same effect as a user-defined constructor with empty body and empty initializer list.
Implicitly-defined copy constructor	Implicitly-defined copy constructor of a class type (class or struct) performs full member-wise copy of the object's bases and non-static data members, in their initialization order, using direct initialization.
Implicitly-defined move constructor	Implicitly-defined move constructor of a class type (class or struct) performs full member-wise move of the object's bases and non-static members, in their initialization order, using direct initialization with an xvalue argument.
Implicitly-defined copy assignment operator	Implicitly-defined copy assignment operator of a class type (class or struct) performs full member-wise copy assignment of the object's bases and non-static data members, in their initialization order, using built-in assignment for the scalars and copy assignment operator for class types.
Implicitly-defined move assignment operator	Implicitly-defined move assignment operator of a class type (class or struct) performs full member-wise move assignment of the object's direct bases and immediate non-static data members, in their initialization order, using built-in assignment for the scalars, member-wise move-assignment for arrays, and move assignment operator for class types (called non-virtually).

Implicitly-defined destructor	Implicitly-defined destructor has an empty body. After the body of the destructor is executed, the destructors for all non-static non-variant data members of the class are called, in reverse order of declaration. Then it calls destructors of all direct non-virtual base classes, in reverse order of construction, and then it calls the destructors of all virtual bases.
Is-a relationship	Subsumption relationship between types. If one class B is a subclass of another class A (i.e. B is a more specialized concept than A), then B class specification implies A class specification and a B class object can be used for any expression that requires an A class object.
Has-a relationship	Composition relationship where one object is a part or member of another object with respect to the rules of ownership.

Table C.1: Acronyms

Definition:	**Description:**
Single point of exit	Approach background: • IEC 61508 [19], as one of methods for providing modular approach • ISO26262 part 6 [5] with the requirement for ASIL A-D. • MISRA-C++ 2008 with the rule M6-6-5. AUTOSAR Coding Guidelines consider that the only reason for such an approach is improving robustness of resource handling, e.g. ensuring that resources are properly released in case of an early exit from the function. However, it is fully ensured by other rules existing in the document that: • enforce usage of smart pointers and memory management objects for expressing lifetime semantics (A18-5-2) • enforce allocate local objects on stack (A18-5-8) • recommend usage of RAII (A15-1-4, A18-5-2) Single point of exit is considered to decrease code readability and will not bring any additional benefits for improving coding standards, thus it is not enforced by the AUTOSAR Coding Guidelines.

Table C.2: Definitions

D Changelog

This section shows changes done between document releases.

D.1 Release 17-10

Type of change:	Modified rules:
Title, example, exceptions	A7-1-7, A15-0-4, A15-0-5, A15-3-1, A18-5-2
Rule classification	A9-3-1
References	A0-4-3, M3-4-1, M5-2-12, A5-0-1, A5-1-2, A5-1-4, A6-5-2, M7-1-2, M7-5-2, A7-1-1, A7-1-7, A8-4-1, M9-3-3, A12-0-1, A12-8-6, A18-5-2, A18-9-2
New rule	A0-1-4, A0-1-5, A6-5-3, A8-4-4, A9-5-1, A12-1-5, A12-1-6, A13-5-2, A18-1-6, A18-5-8
MISRA review changes	A0-1-3, A2-10-5, A5-1-7, M10-1-2
Other	Traceability updated for HIC (see B.2), CERT (see B.4), C++ Core Guidelines (see B.5). Added changelog appendix chapter.

Table D.1: Changelog for release 17-10.

D.2 Release 18-03

Type of change:	Modified rules:
Title, example, exceptions	A1-4-1, A2-3-1, A2-7-3, A2-8-1, A3-3-2, A5-2-5, A12-0-1, A12-8-1, A13-3-1, A15-0-5, A15-3-2, A15-3-3, A15-4-5, A15-5-1, A18-0-2, A18-1-1, A18-1-4, A18-1-6, A18-5-8
Rule classification, numbering	A5-6-1, A2-13-1, A2-13-2, A2-13-3, 6.2.5 (A2-5-2), 6.2.8 (A2-8-1) 6.2.10 (A2-10-1, A2-10-4, A2-10-5), A15-0-6, A15-0-7, A15-0-8, A15-1-1, A15-3-3, A18-1-1
Chapter numbering	6.2.3 (A2-3-1), 6.2.5 (A2-5-2), 6.2.8 (A2-8-1) 6.2.10 (A2-10-1, A2-10-4, A2-10-5)
References, notes	M0-1-9, A2-7-3, A2-8-1, A3-3-2, A4-7-1, A5-0-1, M5-0-2, A5-2-1, A5-2-5, M5-2-2, A6-5-1, A7-1-7, A8-5-2, A9-3-1, A12-1-1, A12-4-1, A12-4-2, A13-5-2, A14-1-1, A15-0-4, A15-0-5, A15-4-1, A15-5-2, A15-5-3, M18-0-4, M18-0-5, A18-5-5

New rule	A0-1-6, A0-4-4, A1-4-3, M2-7-1, A2-7-5, A2-8-2, A2-10-6, A2-11-1, A2-13-4, A2-13-5, A2-13-6, A3-1-5, A3-1-6, A3-8-1, A5-0-4, A5-1-9, A5-2-6, A5-3-2, A5-3-3, A5-5-1, A6-2-1, A6-5-4, A7-1-9, A7-2-5, A7-3-1, A7-6-1, A8-4-5, A8-4-6, A8-4-8, A8-4-9, A8-4-10, A8-4-11, A8-4-12, A8-4-13, A8-5-0, A10-0-1, A10-0-2, A10-4-1, A12-0-2, A13-5-3, A13-5-4, A14-5-1, A14-7-2, A14-8-2, A17-6-1, A18-5-9, A18-5-10, A20-8-1, A20-8-2, A20-8-3, A20-8-4, A20-8-5, A20-8-6, A20-8-7, A21-8-1, A23-0-2, A25-1-1, A25-4-1, A26-5-1, A26-5-2, A27-0-3, A27-0-4
Removed rule	M0-1-5, A1-4-2, A2-7-4, A2-10-2, A2-10-3, M2-10-6, A5-1-5, M5-2-1, M7-3-5, M8-5-1, A13-1-1, M14-5-2, M14-7-3, A14-8-1, M14-8-1, A15-3-1, A15-4-6, A18-1-5
Definitions	`Is-a-relationship`, `Has-a-relationship`
Other	Traceability updated for MISRA (see B.1), HIC++ (see B.2), JSF (see B.3), CERT (see B.4), C++ Core Guidelines (see B.5)

Table D.2: Changelog for release 18-03.

D.3 Release 18-10

Type of change:	**Modified rules:**
Title, example, exceptions	A8-5-2, A8-5-4, A9-6-1, A10-2-1, A11-3-1, A13-5-4, A15-4-3, A16-2-2, A16-2-3, A18-5-2, A18-5-3
References, notes	A5-0-4
New rule	A6-2-2, A8-4-14, M9-6-4, A13-5-5, A14-5-2, A14-5-3, A18-5-11
Other	Traceability updated for MISRA (see B.1), C++ Core Guidelines (see B.5), HIC++ (see B.2). Traceability added for ISO 26262 (see B.6).

Table D.3: Changelog for release 18-10.